

AD-A183 420

①
DTIC FILE COPY

The New ROSIE[®] Reference Manual and User's Guide

James R. Kipps, Bruce Florman, Henry A. Sowizral

This document has been approved
for public release and sale; its
distribution is unlimited.

DTIC
ELECTE
AUG 11 1987
A

RAND

NATIONAL DEFENSE
RESEARCH INSTITUTE

87 8 11 047

①
R-3448-DARPA/RC

The New ROSIE[®] Reference Manual and User's Guide

James R. Kipps, Bruce Florman, Henry A. Sowizral

June 1987

Prepared for the
Defense Advanced Research Projects Agency

RAND

DTIC
SELECTE
AUG 11 1987
S A D

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER R-3448-DARPA/RC	2. GOVT ACCESSION NO. ADA183470	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The New ROSIE Reference Manual and User's Guide		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) J. Kipps, B. Florman, H. Sowizral		8. CONTRACT OR GRANT NUMBER(s) MDA903-85-C-0030
9. PERFORMING ORGANIZATION NAME AND ADDRESS The RAND Corporation 1700 Main Street Santa Monica, CA 90406		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Agency Department of Defense Arlington, VA 22209		12. REPORT DATE June 1987
		13. NUMBER OF PAGES 372
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for Public Release - Distribution Unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) No Restrictions		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Programming Languages Programming Manuals		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) see reverse side		

ROSIE (Rule Oriented System for Implementing Expertise) is a computer programming language/environment developed for the Defense Advanced Research Projects Agency. ROSIE is designed specifically for developing expert systems, and its primary purpose is to aid the knowledge acquisition process. To this end, ROSIE assumes its most characteristic feature, an expressive and highly readable English-like syntax. This report constitutes a new reference manual and user's guide for the ROSIE programming language/environment. It consists of an informal yet detailed discussion of the syntax and semantics of ROSIE 3.0, including an explanation of the programming environment as a whole. The report supersedes all earlier documents describing ROSIE. A technical audience is assumed, but the introduction provides a comprehensive overview of the ROSIE language for those interested in a less technical presentation.

Keywords:
data bases; input/output;
computer files; debugging.

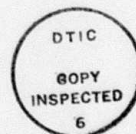
PREFACE

ROSIE¹ (Rule-Oriented System for Implementing Expertise) is a computer programming language/environment developed at The RAND Corporation for the Defense Advanced Research Projects Agency. ROSIE evolved from a succession of projects in artificial intelligence and expert systems. The most recent release of ROSIE (Version 3.0) is the result of an effort to refine and enhance the fundamental design of the language; while all extent versions of ROSIE are implemented in Interlisp, ROSIE 3.0 is implemented in PSL (Portable Standard Lisp). Many aspects of ROSIE 3.0 show performance improvements of two to five times over earlier versions. In addition, ROSIE 3.0 can be ported to any computing environment capable of supporting PSL. Because PSL is supported on a wider range of computing environments than Interlisp, the availability of ROSIE has increased substantially with the 3.0 release.

This report constitutes a new reference manual and user's guide for the ROSIE programming language/environment. It consists of an informal yet detailed discussion of the syntax and semantics of ROSIE 3.0, including an explanation of the programming environment as a whole. The report supersedes all earlier documents describing ROSIE, including RAND Notes N-1648-ARPA, *Rationale and Motivation for ROSIE*, November 1981; N-1647-ARPA, *The ROSIE Language Reference Manual*, December 1981; and N-1646-ARPA, *Programming in ROSIE: An Introduction by Means of Examples*, February 1982, all of which describe ROSIE 1.0. The present report also supersedes RAND Report R-3246-ARPA, *ROSIE: A Programming Environment for Expert Systems*, October 1985, which describes ROSIE 2.5.

This study was prepared for the Defense Advanced Research Projects Agency under RAND's National Defense Research Institute, a Federally Funded Research and Development Center supported by the Office of the Secretary of Defense. In addition, RAND's own research funds were used to complete the effort.

¹ROSIE is a registered trademark of The RAND Corporation.



Accession For	
TIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

SUMMARY

This report is a reference manual and user's guide for the ROSIE programming language/environment and is intended to serve as the primary documentation for this system. The main body of this report consists of an informal, yet detailed, discussion of the syntax and semantics of ROSIE (Version 3.0), including an explanation of the programming environment as a whole. A technical audience of readers is assumed; readers interested in a less-involved presentation are directed to the introduction of this report, which provides a comprehensive overview of the ROSIE language.

ROSIE (Rule-Oriented System for Implementing Expertise) is a general-purpose programming language/environment, designed specifically for developing expert systems. The language has evolved from a relatively simple initial design (Waterman et al., 1979) to a sophisticated expert system building tool (Sowizral and Kipps, 1985). The culminating effort of the ROSIE Language Development Project is ROSIE 3.0, in which the fundamental design of the language has been refined and enhanced.

The primary motivation behind ROSIE's design is to aid the knowledge acquisition process (i.e., the process by which knowledge engineers formalize the heuristics of an expert into executable code). To this end, ROSIE assumes its most characteristic feature, an expressive and highly readable English-like syntax. A second objective has been to support the development of significant applications. ROSIE provides a variety of language and programming environment features aimed at this goal. The language allows system builders to describe complex relations simply and to manipulate them symbolically and deductively.

The ROSIE project was initiated in 1979; the first release of the language, ROSIE 1.0, was available by 1980. Since then, the language has evolved through many phases of design and refinement. The current release, ROSIE 3.0, reflects insights gained from several years of experience with large ROSIE applications. ROSIE 3.0 is substantially more powerful and less constrained in both its syntax and semantics than any previous release.

The work presented here is not merely a concatenation or recompilation of existing documents, but rather it is a newly written document providing an informal description of the ROSIE language. In it, all aspects of the language that concern the ROSIE programmer are discussed in detail, including several previously undocumented and unsupported features of the language. Incompatibilities with older releases are pointed out where applicable.

OVERVIEW

This report is a reference manual and user's guide for the ROSIE programming language and development environment, describing the version 3.0 release of ROSIE. This document supersedes all other manuals and documents describing ROSIE; notably, Hayes-Roth et al., 1981, Fain et al., 1981, and Fain et al., 1982, each of which describe the earlier 1.0 release, and Sowizral and Kipps, 1985, which describes the 2.5 release.

WHAT IS ROSIE?

ROSIE (Rule-Oriented System for Implementing Expertise) is a general-purpose programming language/environment intended for applications in Artificial Intelligence (AI). In particular, The RAND Corporation developed ROSIE to provide knowledge engineers with an environment for building expert systems. ROSIE attempts to provide a complete working environment for developers of expert systems. System builders may create and maintain program files, execute and edit code, and manipulate databases all from within ROSIE's interactive environment.

The ROSIE language itself supports a number of advanced programming capabilities, some of which can be found in other high-level AI programming languages, others of which are unique to ROSIE. These features include:

- rulesets to modularize and scope rules, localizing the context in which they apply
- a demon facility to provide event-driven program control
- high-level data types for manipulating units of procedural, declarative, and descriptive knowledge as data
- a string pattern matcher to support advanced I/O operations
- extended variations of the data types and control structures found in most symbolic languages

Features such as rulesets and the pattern matcher blend with the naturalness of ROSIE's English-like syntax to produce a comfortable and expressive environment in which to construct expert rules.

ROSIE is designed to be adaptable to a wide variety of tasks and does not embody any particular problem-solving techniques or paradigms. Because of its "general-purpose" flavor, it is less structured and more flexible than many contemporary AI systems and tools. Nonetheless, the design of ROSIE exploits and integrates many current ideas in artificial

intelligence research and brings substantial modeling capabilities into the hands of expert system development teams.

READABILITY

The primary design goal for the ROSIE language is that it achieve exceptional readability. To this end, ROSIE adopts a stylized version of English as its syntax. This goal derives from a motivation to aid the knowledge acquisition process (i.e., the process by which knowledge engineers formalize the heuristics of an expert into an executable program). ROSIE's English-like syntax is its most characteristic feature, lending the language a number of desirable qualities. It enables ROSIE rules to be understood by nonprogrammers and assists the knowledge engineer in constructing computational models by providing a framework that is adaptable to a wide variety of problem domains.

With ROSIE, the knowledge engineer can translate expert rules into an executable program using substantially the same terminology as the domain expert. As a result, ROSIE's English-like syntax improves the interaction between those involved in the knowledge acquisition process. The domain expert can examine the heuristics encoded in ROSIE, suggest modifications directly, and play a greater role in the implementation of systems that model his expertise. Also, because programs can be scanned and modified interactively, heuristic development is made possible (and even practical) in a conference or demonstration environment, without the usual delays associated with program modification.

These benefits, however, are not automatic. Although readable code is not hard to generate for an experienced ROSIE programmer, novices often find that it is just as easy to write cryptically as clearly--sometimes even easier. Also, syntax can be a very superficial artifact of any programming language; when the syntax mimics English, the possibilities for misinterpretation are only increased. Although ROSIE gives the illusion of understanding a rich subset of the English language, ROSIE is *not* a natural-language understanding system. ROSIE parses a program by concentrating on the lexical and grammatical role of words as they relate to one another in legal sentence forms. The parsing process strictly follows the syntactic rules defined in a contrived context-free grammar. Thus, the semantic meaning of ROSIE code can at times be counterintuitive to its natural interpretation.

HISTORY OF DEVELOPMENT

As with most other computer systems reaching a relatively mature level of documentation, ROSIE has a long history of development. ROSIE has been an ongoing research effort at RAND since 1979; the language has grown and evolved in many ways over the years. We have worked at improving ROSIE's expressive power without sacrificing its readability, at regularizing its grammar without sacrificing its expressiveness, and at extending its semantics without introducing new complexities.

The historic precursor to ROSIE is the RAND Intelligent Terminal Agent, RITA (Anderson et al., 1977; Anderson and Gillogly, 1976). Influenced by the success of MYCIN's rule-oriented style of knowledge representation and the appeal that its English-like explanation facility had for users, RITA was a positive first attempt at making rule-based programming languages easier to use and understand. Production rules in RITA were defined using an English-like syntax with a restricted set of options. RITA's database consisted of object/attribute/value triples, and its monitors allowed either pattern-directed or goal-directed control. Although its syntactic and expressive powers were limited, RITA showed that a stylized form of English could be used for describing procedural knowledge in a rule-based language.

The preliminary design of ROSIE (Waterman et al., 1979), developed by Donald Waterman, Fredrick Hayes-Roth, Robert Anderson, Stanley Rosenschein, Gary Martins, and Philip Klahr, proposed a programming system that would be the natural successor to RITA. The proposal outlined the deficiencies in RITA and described how they might be overcome in ROSIE. One such deficiency noted was the awkward manner in which context switching was achieved in RITA (i.e., scoping the applicability of rules); this observation influenced the introduction of rulesets as a programming construct in ROSIE. ROSIE also adopted several of the best ideas from RITA, such as RITA's I/O pattern matcher, and improved upon RITA's expressiveness and semantics. The first implementation of ROSIE (Version 0) was written by Danny Gorlin using Interlisp on a PDP-10 class computer and, later, on a DECsystem-20/60. Stanley Rosenschein served as project leader for the initial development effort, and introduced some prolog-like constructs and English-like descriptions into the language.

The second implementation of ROSIE (Version 1) was heavily influenced by the insights gained from the 1980 Expert Systems Workshop organized in part by Fredrick Hayes-Roth and Donald Waterman. Stanley Rosenschein left RAND in 1980, passing leadership of the project to Fredrick Hayes-Roth. Henry Sowizral joined RAND about this time and took over ROSIE implementation after Danny Gorlin left. In 1981, the design of ROSIE had begun to reach a level of stability. A number of in-house applications were being explored using ROSIE, and copies of ROSIE 1.0 were being distributed to sites outside of RAND. Keith Westcourt and Jill Fain joined the project in 1981. Keith Westcourt implemented the port facility for communicating with TOPS-20 from inside ROSIE and reimplemented ROSIE's string pattern matcher. Jill Fain helped test and document the language.

These early implementations of ROSIE included direct support for many special-purpose operations. Such operations were hardwired into the language because they did not fit easily into any general linguistic structure. Some operations required special arguments, others performed actions that were considered expedient in a programming language. As the number of special action verbs began to multiply, ROSIE's grammar grew increasingly complex, and the need to simplify became

overwhelmingly apparent. From 1982 to 1986, generalized linguistic constructs were introduced that could subsume many of the existing "special" constructs. Two examples include removing distinctions between system-defined and user-defined operations and introducing new data elements, such as patterns and filesegments, to describe hitherto special argument forms.

Henry Sowizral became project leader in 1982 when Fredrick Hayes-Roth left RAND. This was the beginning of the third implementation of ROSIE (Version 2), in which the development environment was moved from Interlisp under TOPS-20 to VAX-Interlisp on the VAX 11/780 and Interlisp-D on the XEROX-1100 (Dolphin). Keith Westcourt and Jill Fain left RAND later that year, after which James Kipps came to RAND and took over ROSIE implementation. Ross Quinlan, also at RAND that year, implemented an efficient context-free parser generator in C that replaced ROSIE's LISP-based parser on the VAX implementation.

In 1983, ROSIE 2.3 was released, and work was begun on a redesign of the language aimed at expanding its functionality and improving its performance. Two parallel efforts started at this time were the reimplementing of ROSIE in PSL (Portable Standard Lisp) and the development of a ROSIE compiler in C. Jed Marti and Larry Baer joined the project in 1983; Jed Marti, to automate the porting of ROSIE's Interlisp sources to PSL, and Larry Baer, to implement C-ROSIE. Unfortunately, neither effort bore fruit. During this time, James Kipps implemented two experimental versions of ROSIE: D-ROSIE, a distributed language, in which ROSIEs running on remote machines (Dolphins) communicated via shared databases; and XPLROSIE (Waterman et al., 1986), which provided built-in explanation facilities to support the explanation research being conducted by Donald Waterman and Jody Paul. Bruce Florman, who joined RAND in 1984, took over the further development of XPLROSIE.

The final implementation of ROSIE (Version 3.0) was completed in 1985 by James Kipps and Bruce Florman. ROSIE 3.0, essentially a redesign and reimplementing of the ROSIE language, is written in Portable Standard Lisp and is documented by this report. An effort outside of RAND is currently continuing the development of a C-based version of ROSIE.

The ROSIE Language Development Project culminated its efforts in December of 1985. DARPA funding for the ROSIE project was discontinued after 1985, but the project was given an additional year of funding by RAND, with James Kipps as project leader, to support distribution as well as to continue testing and debugging. Although the ROSIE project has run to completion, the ROSIE environment is still in use at RAND and elsewhere, providing a testbed for new research.

NOTATIONAL CONVENTIONS

The following notational conventions hold when syntactic constructs and examples of their use are presented and discussed:

Boldface Examples of ROSIE code appear in boldface. Boldface is also used to highlight fixed parts of language constructs when appropriate.

Italics Italicized words either designate fundamental data types, i.e.,

a name
a number
a string
a pattern
a tuple
a filesegment
a class element
a description
a proposition
a procedure

or a role that such elements can play, i.e.,

a file
a database

When appearing as an argument to a syntactic construct, e.g.,

send *a string* to *a file*

they specify that any term can be used as an argument in that position, but that term *must* evaluate to an element which satisfies given type or role.

<standard> Words in standard font and enclosed in matching left/right angle brackets represent syntactic categories. When appearing in a linguistic construct, e.g.,

<term> DOES <atom> [<term>] [<pphrase>]

they designate its component syntactic structure.

The following syntactic categories appear throughout the manual:

<atom> -- any nonreserved atomic token

<number> -- any numeric token (real or integer)

<integer> -- any positive integer

<string> -- any sequence of characters surrounded by double quotes ("ccc")

<prep> -- any token known to be a preposition (see Section 2.3)

<a/an> -- one of the tokens **a** or **an**

<term> -- any *term*, i.e., high-level data object, (see Chapter 8)

<pphrase> -- a sequence of prepositional phrases, i.e.,

<pphrase> ::= [<prep> <term>]*

A complete description of ROSIE's lexical and syntactic components appears in Section 2.4.

- [] Constructs enclosed by matching square brackets are optional.
- []* Square brackets followed by a star indicate *Kleene closure*, i.e., the enclosed constructs can appear zero or more times.
- (| |) Parentheses and vertical bars surround alternatives separated by vertical bars, only one of which can occur in that position.

Italics are used to highlight new concepts when they are introduced.

When discussing programming operations available in the basic configuration of the ROSIE system, the following notational conventions apply. Within this notation, words in angle brackets correspond to fixed parts of the construct being discussed--these will appear in boldface in the manual--while words in italics correspond to variable parts of the construct--these will appear in italics.

<imperative> [*a/an argtype*] [<prep> *a/an argtype*]*

Used to define the syntax of system defined procedures, where <imperative> is the name of the procedure, and the optional object and <prep> clauses correspond to possible arguments to the procedure, e.g.,

read *a pattern* [**from** *a file*]

a/an argtype <verb> [*a/an argtype*] [<prep> *a/an argtype*]*

Used to define the syntax of system predicates, where <verb> corresponds to the auxiliary and main verb of the predicate,

and the subject, optional object, and <prep> clauses correspond to possible arguments to the predicate, e.g.,

a string is matched by a pattern

the <class> [<prep> *a/an* argtype]*
a/an <class> [<prep> *a/an* argtype]*

Used to define the syntax of system generators, where <class> corresponds to the root name of the class being generated over, and the optional <prep> clauses correspond to possible arguments to the generator.

Note the two forms this can take: One is introduced with the articles *a/an*; and the other with the article *the*. This notation is used to signify whether the generator produces a single instance, e.g.,

the absolute value of a number

or a possible stream of instances, e.g.,

an integer from a lower bound to an upper bound

STRUCTURE OF MANUAL

In structuring this manual, we were faced with several competing goals. First, we wanted this to be the primary source of documentation for the ROSIE language. Next, we wanted to provide system builders with sufficient information to allow them to use the language well. Finally, we wished to provide an informal, yet comprehensive, set of implementation criteria to sustain those who support and maintain ROSIE in the future.

Meeting our first objective required that we make the information within this manual available to a wide audience of readers. Thus, it had to be complete and concise without being overly detailed. Obviously, this would be counter to our second two objectives, which require a detailed and precise description of the language and assume a technical audience. In order that we might meet both goals, we provide a comprehensive, but nontechnical, overview of ROSIE in the introduction, and then assume a technique audience of readers for the remaining chapters.

Chapter 1, the introduction, gives new ROSIE users a quick look at the language and some familiarity with concepts appearing later in the manual. The introduction is also recommended reading for those with prior ROSIE experience since it may offer some advance warning to the changes and additions to be found in ROSIE 3.0. Readers who merely wish to know what ROSIE is and what it can do are directed to the introduction and Appendix A, which presents several example programs.

Chapter 2 describes the syntactic structure of the language. In this chapter, we discuss the lexical decomposition that ROSIE source code goes through on its way to the parser. We explain how the parser operates and provide a listing of the BNF describing ROSIE's syntax as well as the reserved words of the language.

Chapter 3 discusses the basics of running ROSIE. This chapter is intended to provide new users with the information they need to start interacting with ROSIE as well as building ROSIE programs.

The next six chapters constitute the main body of text and describe the actual structure of the language. When pertinent, sections within these chapters begin with an abridged portion of the BNF demonstrating the syntax of the particular linguistic components being presented. Also when pertinent, these chapters include a list of operations provided to work on these components.

At first glance, the organization of these chapters may seem counterintuitive. First we present the programming structures (*rules* and *rulesets*), then the principle linguistic structures (*actions*, *sentences* and *descriptions*), and finally the data objects and data primitives (*terms* and *elements*). Readers already familiar with ROSIE or other high-level programming languages may question this scheme because it is the reverse order in which such concepts are normally presented. We feel, however, that this is the correct approach for properly explaining ROSIE.

ROSIE is unlike other programming languages in that the data types it provides are not its simplest component. While ROSIE supports extensions of the data types commonly found in other symbolic languages, it also supports some advanced data types, such as the intentional elements, which are used to treat principal linguistic structures as data. To discuss such data types, we must first introduce actions, sentences, and descriptions. However, these structures often invoke rulesets, and understanding their semantics assumes an understanding of rulesets and ruleset invocation; thus, a discussion of rulesets must precede them. However, a major component of ruleset invocation is the execution of rules. This finally led to our ordering our discussion accordingly as *rules*, *rulesets*, *actions*, *sentences*, *descriptions*, *terms*, and *elements*.

Chapter 10 describes the structure of ROSIE's database, which, up to this point, has only been alluded to. The database is composed of two spaces: one, the space of stored or *affirmed* relations; the other, the space of computed or *virtual* relations.

Chapters 11 and *12* cover ROSIE's support of input and output (I/O) and the error handler. In Chapter 11, we discuss the basic sorts of I/O operations provided in the language. In Chapter 12, we talk about unrecoverable and recoverable runtime errors and what mechanism ROSIE provides for dealing with them.

Chapters 13 and 14 discuss two important features of ROSIE's development environment, the file package and the break package. The file package is used to build, edit, and otherwise maintain program files. The break package is used to monitor and debug program behavior.

This manual contains three appendixes. *Appendix A* presents several example programs. *Appendix B* is a listing of possible error messages to be encountered when parsing or running a ROSIE program. *Appendix C* specifies the various system switches available for configuring various aspects of the ROSIE environment.

ACKNOWLEDGMENTS

We would like to thank Jean Thomas for her help in preparing this report for publication, and Robert Weissler and Jody Paul for their insightful and cogent reviews. We would also like to thank Joyce Grey, who administers the distribution of ROSIE. But we would especially like to acknowledge the efforts of the many people who influenced the evolution of the ROSIE language, including Robert Anderson, Larry Baer, Jill Fain, Fredrick Hayes-Roth, Daniel Gorlin, Philip Klahr, Jed Marti, Gary Martins, Jody Paul, Ross Quinlan, Stanley Rosenschein, Donald Waterman, and Keith Westcourt; a summary of their contributions can be found in the Overview under "History of Development."

CONTENTS

PREFACE	iii
SUMMARY	v
OVERVIEW	vii
What Is ROSIE?	vii
Readability	viii
History of Development	viii
Notational Conventions	xi
Structure of Manual	xiii
ACKNOWLEDGMENTS	xvii

Section

I. INTRODUCTION	1
1.1 System Organization	1
1.2 Programming Structures	2
1.2.1 Rules	2
1.2.2 Rulesets	3
1.2.3 Demons	4
1.2.4 Ruleset Execution	5
1.3 Linguistic Structures	6
1.3.1 Actions	6
1.3.2 Sentences	6
1.3.3 Terms	7
1.3.4 Elements	8
1.4 The Database Mechanism	9
1.4.1 The Physical Database	9
1.4.2 The Virtual Database	10
1.5 Strengths and Weaknesses	10
1.6 Closing Remarks	12
II. SYNTACTIC STRUCTURE	15
2.1 Parsing Basics	15
2.2 Tokenization	17
2.2.1 The Character Set	18
2.2.2 Tokens and File Items	20
2.2.3 Comments	22
2.2.4 Extended String Syntax	23

2.3	Reserved Words	26
2.4	The ROSIE Grammar	27
2.4.1	The Lexical BNF	27
2.4.2	The Linguistic BNF	28
2.5	Parse Tree Generation	38
2.5.1	Associativity, Precedence, and Disambiguation ...	38
2.5.2	The Disambiguation of Prepositional Phrases	39
2.5.3	Directing Disambiguation with Parentheses	40
III.	RUNNING ROSIE	43
3.1	Getting Started	43
3.2	Interactions at the Top Level	44
3.3	Building ROSIE Programs	48
3.4	Debugging Facilities	53
3.5	Errors, Interrupts, and Break Loops	56
3.6	Exiting a ROSIE Session	58
3.7	System Switches	58
3.8	Top-Level Operations	59
IV.	PROGRAMMING STRUCTURES	67
4.1	Rules	67
4.2	Rulesets	68
4.2.1	Defining Rulesets	69
4.2.1.1	Header Statements	70
4.2.1.2	Private Class Declarations	72
4.2.1.3	Execution Monitors	72
4.2.1.4	The Ruleset Body	73
4.2.1.5	End Statements	73
4.2.2	Ruleset Types	73
4.2.2.1	Procedural Rulesets	73
4.2.2.2	Predicate Rulesets	74
4.2.2.3	Generator Rulesets	76
4.2.3	Invoking Rulesets	77
4.2.3.1	Calling Forms	77
4.2.3.2	Argument Passing	78
4.2.3.3	The Private Database	78
4.2.3.4	Execution Monitors	79
4.2.3.5	Terminating Procedures	79
4.3	Demons	81
4.3.1	Types of Demons	82
4.3.2	Demon Invocation	83
4.3.3	The Generator Demons	83
4.3.4	The Error Demon	85
4.4	System Rulesets	85
4.4.1	Defining System Rulesets	86
4.4.2	Calling System Rulesets	86

V. ACTIONS AND CONTROL FLOW	89
5.1 Actions and Action Blocks	89
5.1.1 Types of Actions	89
5.1.2 Associativity of Action Blocks	91
5.1.3 Comma Blocks and Parentheses	91
5.2 Procedures	93
5.3 Database Actions	93
5.3.1 ASSERT... and DENY...	94
5.3.2 LET...	94
5.3.3 CREATE...	95
5.4 Conditional Actions	95
5.4.1 IF... and UNLESS...	96
5.4.2 Associativity	96
5.5 Conditional Blocks	96
5.5.1 SELECT...	97
5.5.2 CHOOSE...	97
5.5.3 MATCH...	98
5.5.4 Associativity	98
5.6 Iterative Actions	100
5.6.1 FOR EACH...	101
5.6.2 WHILE... and UNTIL...	101
5.6.3 Associativity	102
VI. CONDITIONS, SENTENCES, AND PROPOSITIONS	105
6.1 Conditions and Boolean Connectors	105
6.1.1 Boolean Connectors	105
6.1.2 Associativity and Precedence	106
6.2 Sentences	106
6.2.1 Propositions	107
6.2.1.1 Verb Phrases	108
6.2.1.2 Properties of Class Relations	109
6.2.1.3 Negation: NOT...	110
6.2.2 Special Sentence Forms	110
6.2.2.1 EQUAL TO..., LESS THAN..., and GREATER THAN...	111
6.2.2.2 THERE IS...	112
6.2.2.3 HAS...	112
VII. DESCRIPTIONS AND CLASSES	115
7.1 Classes	116
7.1.1 Testing for Membership	116
7.1.2 Generating from a Class	118
7.1.3 Potential Pitfall to Class Membership	119
7.2 Relative Clauses	120
7.2.1 Logical Groupings	121
7.2.2 SUCH THAT... and WHERE...	122
7.2.3 THAT..., WHICH..., and WHO...	123

7.2.4 WHOSE...	124
7.2.5 WHICH... and WHOM...	124
7.2.6 EXCEPT...	125
7.3 Description Variables	125
7.3.1 Anaphoric Terms and Rule Variables	126
7.4 Anaphoric Descriptions: SUCH...	127
7.5 Resolving Anaphoric References	128
7.6 Uses of Descriptions	128
7.6.1 Testing for Membership	129
7.6.2 Generating Elements	129
7.6.3 Asserting and Denying Members	130
7.7 Compound Classes versus Adjectives	131
VIII. TERMS	133
8.1 Types of Terms	133
8.2 Elements	134
8.3 Arithmetic Expressions	135
8.3.1 Operators and Operations	135
8.3.2 Associativity and Precedence	137
8.4 Descriptive Terms	137
8.4.1 Simple Descriptive Terms	138
8.4.1.1 THE...	138
8.4.1.2 A... and AN...	140
8.4.1.3 A NEW...	140
8.4.2 Quantified Descriptive Terms	141
8.4.2.1 SOME...	142
8.4.2.2 EVERY...	143
8.5 Anaphoric Terms and Rule Variables	144
8.6 Iterative Terms	145
8.6.1 ONE OF... and EITHER...	146
8.6.2 EACH OF... and BOTH...	147
IX. ELEMENTS	149
9.1 Element Basics	149
9.1.1 Types of Elements	149
9.1.2 Evaluation Names	151
9.1.3 Equivalence versus Equality	151
9.1.4 General Operations on Elements	153
9.2 Names	159
9.3 Numbers	161
9.3.1 Types of Numbers	161
9.3.2 Constraints on Numbers	163
9.3.3 Operations on Numbers	163
9.4 Tuples	167
9.4.1 Operations on Tuples	167
9.5 Strings	171
9.5.1 Formatted Strings	171

9.5.2 Strings and Patterns	172
9.5.3 Extended String Syntax	173
9.5.4 Operations on Strings	173
9.6 Patterns	177
9.6.1 Generating Text	177
9.6.2 Matching Text ..	178
9.6.3 Subpatterns	179
9.6.4 Pattern Variable Binding	191
9.6.4.1 Pattern Variable Specification	192
9.6.4.2 Conversion of Bound Substrings	193
9.6.5 The Pattern Matching Process	193
9.6.6 Example Application of Patterns	195
9.6.7 Operations on Patterns	196
9.7 Filesegments	199
9.7.1 Shorthand for Filesegments	199
9.8 Class Elements	201
9.8.1 Motivation and Intended Use	202
9.8.2 Potential Pitfalls	203
9.9 Intentional Descriptions	207
9.9.1 INSTANCE OF...	207
9.9.2 The "Call-by-Name" Property	209
9.9.3 Operations on Intentional Descriptions	209
9.10 Intentional Propositions	211
9.10.1 IS PROVABLY...	211
9.10.2 Operations on Intentional Propositions	212
9.11 Intentional Procedures	217
9.11.1 Operations on Intentional Procedures	217
X. THE DATABASE MECHANISM	219
10.1 The Physical Database	219
10.1.1 Three-Valued Logic	219
10.1.2 Database Actions	220
10.1.3 Contradictory Assertions	220
10.1.4 Alternate Databases	221
10.1.4.1 Naming and Creating Databases	221
10.1.4.2 The Global, Active and Private Databases	221
10.1.4.3 Accessing the Physical Database	223
10.2 The Virtual Database	224
10.2.1 Predicate and Generator Rulesets	225
10.2.2 Virtual Relations	226
10.3 Asserting, Testing, and Denying Propositions	228
10.4 Auto-Query Mode	230
10.5 Database Operations	232
XI. INPUT/OUTPUT	241
11.1 Channels	241
11.1.1 Opening and Closing Channels	241

11.1.2 The Standard I/O and TTY Channels	242
11.1.3 The OS Channel	243
11.2 The Use of Patterns	244
11.2.1 Sending Formatted Text	244
11.2.2 Reading against a Pattern	244
11.3 Creating Transcript Files	245
11.4 Input/Output Operations	246
XII. ERRORS AND ERROR RECOVERY	251
12.1 Nonrecoverable and User Errors	251
12.2 Recoverable Errors	251
12.3 The Error Demon	251
XIII. THE FILE PACKAGE	253
13.1 Program Files	253
13.2 Using the File Package	254
13.3 Defining Rulesets and File Rules	255
13.4 Editing and Modifying Program Files	256
13.5 Using Filesegments	257
13.5.1 Rule Sequence Specifiers	258
13.5.2 Shorthand Notation	259
13.6 File Package Operations	260
XIV. THE BREAK PACKAGE: DEBUGGING PROGRAMS	267
14.1 Breakable Aspects of a Program	267
14.2 The Trace Facility	268
14.3 The Break Facility	270
14.3.1 Break Commands	270
14.3.2 Example Session	274
14.4 The Profile Facility	278
14.5 Restoring Broken Rulesets	279
14.6 Break Package Operations	279
APPENDIX A: EXAMPLE PROGRAMS	283
FORTUNE -- The Basics	285
POIROT -- Alternate Databases	295
ANIMAL -- Embedded Control Structures	305
APPENDIX B: ERROR MESSAGES	313
Parsing and Tokenization Errors	313
Runtime Errors	317

APPENDIX C: SYSTEM SWITCHES	337
Operations on System Switches	342
REFERENCES	345
INDEX	347

I. INTRODUCTION

The introduction is organized in a manner similar to the main body of the manual, though without as much of the laborious detail. This chapter is recommended reading for all new ROSIE users because it provides a broad overview of the language and introduces concepts that will be appearing throughout the manual.

1.1 SYSTEM ORGANIZATION

ROSIE exists as a system of three major components. One component, *the parser*, translates ROSIE source code into a machine-executable representation. *The runtime component* supplies the functions that support the execution of this parsed code. The third component encompasses those features that describe ROSIE's *interactive programming environment*.

Before ROSIE code can be executed, it must be *parsed* (translated) into a machine-executable representation called *HILEV*. Parsing consists of three phases: (1) lexical analysis (or *tokenization*) of the source file (i.e., recognizing *file items*, such as rules and declarations, and transforming each file item into a list of *tokens*); (2) generating a parse tree for each file item; and (3) transforming each parse tree into its executable HILEV representation.

The functions supporting the execution of HILEV fall into three categories: (1) functions that work on *elements* (ROSIE's data primitives); (2) functions that manipulate the database; and (3) functions that invoke rulesets and demons. Functions in the first category define elements and test their equivalence. Elements can be organized into *classes*, so another operation of these functions is to generate instances of a given class and chain through class hierarchies. The database functions provide for such operations as creating databases and switching context between alternate databases. The final set of functions support the invocation of rulesets and demons, which are programming structures used to proceduralize the applicability of rules.

The interactive programming environment provides a workspace for developing, debugging, editing, and running programs. Features include:

- *the top-level monitor*, which accepts and executes commands from the user's terminal
- *the history mechanism*, which allows users to review and re-execute past commands

- *the file package*, which permits users to build, load, examine, edit, and otherwise manipulate program files
- *the break package*, which provides interactive debugging and error recovery mechanisms

ROSIE's working environment shields the user from the complexity of the other two components, essentially acting as a front end to the parsing and runtime components.

1.2 PROGRAMMING STRUCTURES

The principal programming structures in ROSIE are *rules*, *rulesets*, and *demons*. Rules correspond to executable programming statements, while rulesets equate to rule subroutines; demons are a specialized form of ruleset. ROSIE programs are defined as collections of interacting rulesets and demons. To run a program, one issues a rule to ROSIE's top-level monitor, which parses and executes the rule. Supposedly, the rule will invoke a ruleset, which executes the rules in its body, invoking other rulesets, and so on.

1.2.1 Rules

Rules consist of an ordered sequence of *actions*, separated by the conjunctive **and** and terminated by a period (**.**), e.g.,¹

**Assert the report was received at the current time and
relay that report to every module.**

**If any red battalion does advance toward any strategic
objective and that objective is undefended,
move some blue battalion to that objective and
report 'that battalion was directed to that objective'.**

**For each blue battalion (BBTL) in sector #15,
advise BBTL to 'move to Red River Crossing' and
assert BBTL was given a new directive.**

**While any strategic objective is not defended,
keep some blue battalion on alert.**

A rule executes each of its component actions in turn. As one can observe from these examples, ROSIE "rules" differ significantly from the

¹The first example rule contains two actions, an *assert action* and a *procedure*; the second, a *conditional action*; the last two example rules illustrate two different types of *iterative actions*. Note that the conditional and iterative actions take nested action blocks as arguments.

notion of rule found in production system architectures such as OPS5 (Forgy, 1981) and ENYCIN (van Melle, 1981). While ROSIE rules *can* appear in the *if-then* form of production rules, they can also appear using other control abstractions (e.g., as seen above in the *for-each* and *while* forms). The key point to keep in mind is that ROSIE rules are *not* treated like production rules. Rules are treated strictly as executable programming statements.

1.2.2 Rulesets

The applicability and context in which rules are executed can be controlled by organizing rules into rulesets. Like subroutines in more conventional programming languages, rulesets provide a convenient way to modularize rules into coherent procedural units. One of ROSIE's strengths is that these modules can be invoked in a natural and transparent way using generalized English-like linguistic structures. There are three types of rulesets: *procedural*, *predicate*, and *generator* rulesets. Each ruleset type serves a conceptually different purpose; each gets invoked in a different way; and each returns a different form of value.

A procedural ruleset enacts a *procedure* (a type of action) and does not return a result to the calling form. As an example, consider the procedural ruleset,

```
To move a vessel from a source to a destination:  
[1] Deny the vessel is docked at the source.  
[2] Assert the vessel is docked at the destination.  
End.
```

which updates the database when invoked by a procedure such as in

```
Move USS Nimitz from Le Havre to Auckland.
```

A predicate ruleset provides a means of computing the truth or falsity of a *proposition* (a declarative *n*-ary relation). When ROSIE cannot otherwise decide a proposition's *truth value* from relations in its database, it automatically invokes the corresponding predicate ruleset if one exists. For instance, the predicate ruleset

```
To decide if a vessel is seaworthy:  
[1] If the vessel does float, conclude true,  
    otherwise, if the vessel does leak,  
    conclude false.  
End.
```

will be invoked by

```
If USS Nimitz is seaworthy,  
    move USS Nimitz from Le Havre to Auckland.
```

if the proposition 'USS Nimitz is seaworthy' cannot otherwise be proved or disproved from assertions in the database. A predicate ruleset can conclude true or false, returning a boolean value to the calling form, or it can simply terminate, returning nothing and implying an indeterminate truth value.

A generator ruleset produces instances of a *class*. When generating from a class (say, the class of *ship*), ROSIE first produces all elements that satisfy the proposition '*element is a class*' in the database, e.g., assuming the database contains

USS Nimitz is a ship
USS Coral Sea is a ship
USS Enterprise is a ship

then generating **every ship** successively produces **USS Nimitz**, **USS Coral Sea**, and **USS Enterprise**. Once all such elements have been exhausted, ROSIE can invoke a generator ruleset for computing additional members of the class. For instance, the generator ruleset

To generate a vessel at a port:
[1] Produce every boat which is docked at the port.
[2] Produce every ship which is docked at the port.
End.

would produce a continuous stream of elements when invoked by

While some vessel at Auckland is not seaworthy,
repair that vessel.

until all elements produced satisfied the '*element is seaworthy*' predicate. Like predicates, calls to generator rulesets are made transparently through interactions with the database and do not affect the readability of code.

1.2.3 Demons

ROSIE also supports a specialized form of ruleset called *demons*. Demons selectively capture control of computations just prior to the occurrences of an event. Once invoked, a demon can interrogate the system state and either allow the interrupted event to resume or release control without continuing the event. As an example, consider the demon,

Before executing to move a ship from a source to a destination:
[1] Unless some vessel at the source is equal to the ship,
return, otherwise continue.
End.

which would be awakened by

Move USS Nimitz from Le Havre to Auckland.

Execution of the procedure would continue only if its arguments (i.e., **USS Nimitz**, **Le Havre**, and **Auckland**) satisfy the constraints posted by the demon.

Demons provide a mechanism for event-driven program control. They can be used for tracing and debugging during program development. They can monitor changes to the database and check the database for consistency as it undergoes change.

1.2.4 Ruleset Execution

Although rulesets and demons are invoked in different ways, once called they follow the same steps. First, a *private database* is established; this is used to store information that is local to the ruleset invocation, such as parameter bindings. For example, when

To move a ship from a source to a destination:

is invoked by

Move USS Nimitz from Le Havre to Auckland.

the propositions,

**USS Nimitz is a ship
Le Havre is a source
Auckland is a destination**

are asserted into the private database. Next, the rules in the ruleset's body are executed one-by-one according to an *execution monitor*.² There are three types of monitors that execute rules either *sequentially* (first rule to last), *cyclically* (first rule to last then repeat), or *randomly* (any rule at random then repeat). A ruleset invocation terminates when the last rule has been executed (if the monitor is sequential) or when control is explicitly returned to the calling form by a terminating procedure, such as **conclude**, **produce**, **continue**, or **return**.

²The execution monitor should not be confused with the notion of control monitors found in production systems. Although execution monitors "control" the execution of rules, the control is very rigid and does not provide the notion of a *conflict set* or of *conflict resolution* (McDermott and Forgy, 1978).

1.3 LINGUISTIC STRUCTURES

ROSIE supports three fundamental linguistic structures for encoding heuristics: *actions*, *sentences*, and *terms*. Actions advance the flow of control, sentences state declarative relations, and terms function as data objects.

1.3.1 Actions

By definition, a rule always contains at least one action. Actions can invoke procedural rulesets, e.g.,

Deploy a blue battalion to the objective.

conditionally execute an embedded block of actions, e.g.,

**If some red battalion does threaten any strategic
objective and that objective is undefended,
deploy some blue battalion to that objective.**

or iterate over an action block, e.g.,

**For each blue battalion (BBTL) in sector #15,
advise BBTL to 'move to Red River Crossing' and
assert BBTL was given a new directive.**

**While any strategic objective is not defended,
keep every blue battalion on alert.**

Actions such as

**Assert the objective was displayed
Deny battalion #5 is deployed to sector #8
Let the objective be Red River Crossing**

add and remove relations from the database.

1.3.2 Sentences

Sentences specify declarative relations whose truth or falsity can be tested. Some sentence forms test the cardinality of a class, e.g.,

if there is more than one unit which is on alert . . .

while others test the equality of data objects

if the type of aircraft is equal to F-111X . . .

Another type of sentence exists, called a *proposition*, for which truth and falsity can be computed by predicate rulesets.

There are five basic syntactic forms for propositions, each of which captures a specific class of English usage, i.e.,

class membership *term is a description*
battalion #5 is a blue infantry battalion

predication *term is verb [prep term]**
Red River Crossing will be undefended at 0830 hours

predicate complement *term is adjective term [prep term]**
the battalion was deployed rapidly to Red River Crossing

intransitive verbs *term do verb [prep term]**
the battalion will proceed to the objective

transitive verbs *term do verb term [prep term]**
battalion #5 did receive the message at 1500 hours

As the reader may observe from the above examples, propositions may be expressed in either past, present, or future tense³ and modified by prepositional phrases. Propositions can be negated by inserting the word **not** before the main verb, e.g.,

battalion #5 did not receive the message at 1500 hours

The database actions, **assert** and **deny**, take propositions as arguments, adding or removing them from the database. Actions such as **if**, **while**, and **until** accept boolean combinations of sentences, which they test against system state.

1.3.3 Terms

Terms are ROSIE's data objects. They correspond to expressions that have one (or possibly more) values as their interpretation. These values will be one of ten data primitives called *elements*. Terms serve as arguments to actions and sentences as well as other terms.

There are five forms terms can take. *Elements* can act as terms, evaluating to themselves. *Arithmetic expressions*, e.g.,

the DE * (the exposed aircraft / the number of aircraft)

are terms that evaluate to *number elements*. *Descriptive terms*, e.g.,

the distance from home base
every blue battalion which is undeployed

³While ROSIE recognizes a distinction between propositions that vary in tense, it does not use tense information further. Tense is supported merely to allow a wider range of expression.

evaluate to one, some, or all instances of a class.⁴ There are also *anaphoric terms*, e.g.,

that battalion

which evaluate to an element previously produced from a description, and *iterative terms*, e.g.,

**one of F-111X, F-4X or F-16X
each of battalion #5 and battalion #8**

which evaluate to one or all members of a given sequence.

1.3.4 Elements

Terms evaluate to elements. Elements can be divided between two categories: *simple elements* and *intentional elements*. These include:

Simple Elements	Examples
<i>names</i>	battalion #5
<i>numbers</i>	
<i>simple numbers</i>	3.1412
<i>unit constants</i>	55 miles/hour
<i>labeled constants</i>	probability 0.75
<i>strings</i>	"The ratio HEP/COG:"
<i>patterns</i>	{{"Yes"} "No"} (bind to the reply), cr}
<i>tuples</i>	<pol soft, <5 waves, FX-4>>
<i>filesegments</i>	'file: "intel", to report a finding'
 Intentional Elements	
<i>class elements</i>	any non-offensive target
<i>intentional descriptions</i>	'an action at the current time'
<i>intentional propositions</i>	'visibility does approximate 3.5 miles'
<i>intentional procedures</i>	'deploy the unit to sector #3'

Several of the simple elements (i.e., *names*, *numbers*, *strings*, and *tuples*) exist as slightly more complex variations on the basic data types found in most symbolic programming languages. The others (i.e., *patterns* and *filesegments*) provide explicit representation for data structures used in operations that are unique to ROSIE. For instances, *filesegments* identify portions of a program file that can be manipulated via the file package, and *patterns* interface to ROSIE's string pattern matcher and support complex input and output operations.

⁴The major component of a descriptive term is a *description*, which is composed of a class reference (e.g., **blue battalion**) and optionally modified by a relative clause (e.g., **which is undeployed**).

The *intentional elements* provide ROSIE with limited "self-referential" capabilities, allowing programs to treat units of descriptive, declarative, and procedural knowledge as data. *Class elements* and *intentional descriptions* permit program control over the evaluation of class relations, e.g.,

Execute every instance of 'an action at the current time'.

Intentional propositions capture the intent of relations between objects, which can be passed as arguments to rulesets, e.g.,

Report 'visibility does approximate 3.5 miles'.

as well as asserted, tested, or denied. *Intentional procedures* provide a representation for working with suspended actions, e.g.,

Execute 'deploy the unit to sector #3' at time 100.

which can be queued and later executed on demand. Essentially, the intentional elements give knowledge engineers a vehicle for developing meta-level control mechanisms.

1.4 THE DATABASE MECHANISM

The initial, intermediate, and final results of ROSIE programs are stored as *affirmed* propositions in ROSIE's database. Propositions can be asserted (affirmed in the database) and denied (removed from the database). It is possible to test the truth or falsity of a proposition against the contents of the database as well as generate the members of a class defined by affirmed class relations (i.e., propositions using the *is-a* copula).

ROSIE's database structure actually consists of two conceptually separate layers. The first is the *physical database*, which contains affirmed propositions. The second is the *virtual database*, which consists of those relations that can be computed from other relations via ruleset invocation or through a limited deductive retrieval mechanism provided with class elements.

1.4.1 The Physical Database

The physical database can be employed to store facts about the world as well as intermediate computational results. These facts and results must be propositions, which are affirmed using a three-valued logic system. Propositions stored in the database may have a *truth value* of *true* or *false*; propositions that are not in the database have an *indeterminate* truth value. This three-valued logic provides ROSIE with an "open-world" assumption, which implies that ROSIE may not have complete information about a particular situation; truth or falsity will not be inferred in the absence of contradictory information.

Occasionally in AI applications, a method is needed for storing different facts in different databases. This may arise because we wish to model multiple points of view or because we want to restrict attention momentarily to a subset of those facts that are most relevant. To support such needs, ROSIE allows users to create *alternate databases* and specify when they should be *activated* and *deactivated* (i.e., swapped in and out of context).

1.4.2 The Virtual Database

The virtual database supports relations that either cannot be described by affirmed propositions or for which such a representation is not economical. For instance, relations such as '**3 is greater than 2**' are more economical to compute than store. The virtual database consists of both predicate and generator rulesets, and *virtual relations*.

Predicate and generator rulesets allow users to define subroutines for deciding the truth or falsity of a proposition or producing the elements of a class, respectively. Virtual relations, which are affirmed propositions containing a *class element* argument, e.g.,

any ship is a vessel

give users a method for specifying relations that hold over a class of elements.

The essential trade-off between the physical and virtual database is space versus time. In general, relations stored explicitly in the physical database require more memory for their representation than relations stored in the virtual database, while relations stored in the virtual database require extended computation for their retrieval.

1.5 STRENGTHS AND WEAKNESSES

As we have seen, ROSIE programs are described as collections of rulesets and demons that affect the flow of control through direct invocation, initiation of an event, and interactions with the database. ROSIE differs in several respects from other current "expert system" languages. ROSIE is *not* a production system architecture, nor is it an object-oriented or frame-based language or a language for programming logic. ROSIE is similar to production system architectures in that "factual knowledge" is stored as relations in a common database, and all work is done through side effects on these relations. Yet, it is unlike production systems because it does not contain an inference engine, and its "rules" behave like executable code. ROSIE's innate control structure is actually quite similar to the procedural, top-down control structure found in languages such as LISP or PASCAL. However, ROSIE provides a wider array of data abstractions than is available in other procedural programming languages. These aspects distinguish ROSIE from

other contemporary systems and greatly affect how one builds an expert system in ROSIE.

There is a general consensus among those doing research in the area of expert systems that components of knowledge should be abstracted away from components of control (Kowalski, 1979; Hayes-Roth et al., 1983). The strength of production systems rests largely on the distinction they make between knowledge and control. Unfortunately, few production systems provide access to their control mechanisms, and, thus, aspects of control often appear in the knowledge base anyway. In ROSIE, the problems of delimiting knowledge from control are reversed. Since ROSIE resembles a "conventional" programming language, the system builder has greater access to mechanisms for describing control than is available in most production system architectures, which normally provide control abstractions only as some form of inferencing over *if-then* rules. Yet, the system builder has virtually no means of separating aspects of control from knowledge. When building an expert system in ROSIE, one is forced to either intermix control and knowledge or build a control structure on top of ROSIE. Both approaches have the undesirable side effect of detracting from the readability and performance of the resulting system.

The first approach is undesirable because it essentially means that knowledge is "hardwired" into the system. This has the effect of making the system's code very rigid and inflexible. In addition, since no clear delineation of knowledge from control exists, it becomes difficult to distinguish between the two, which detracts from the code's readability for nonprogrammers. The second approach is equally problematic. The essence of this approach is to build a rule interpreter on top of ROSIE--where the interpreter provides a variety of the specialized control structures needed for the target system--and then define the system's knowledge in terms of this new "meta-level" language. The problem with this approach is that ROSIE does not provide hooks into its English-like syntax, nor does it provide a mechanism for easily defining new data types. Thus, knowledge in such a system must be encoded using ROSIE's existing data primitives. While this can be done successfully via the intentional elements, the encoded knowledge loses much if not all claim to being readable. Additionally, the extra level of interpretation substantially degrades system performance.

The lack of support for the abstraction of knowledge from control can be viewed as a fundamental flaw in the design of ROSIE and its major weakness. Despite this, ROSIE does have some redeeming qualities that recommend it for work in expert systems research. ROSIE has been successfully used in the development of several large-scale prototype expert systems. These include: LDS, a Legal Decision-making System, (Waterman and Peterson, 1981); TATR, a Tactical Air Target Recommender, (Callero et al., 1984); DSCAS, a Differing Site Condition Analysis System (Kruppenbacher, 1984); Adept, a workstation to aid combat intelligence analysts (Beebe et al., 1984); and SAL, a System for Asbestos Litigation (Paul et al., 1986).

One of ROSIE's strengths is its use as a prototyping language. As a general-purpose programming language, ROSIE gives the system builder far more "control over control" than is available in most other "expert system" languages. This means that the system builder is not hampered by an otherwise rigid and narrow control structure, allowing the rapid development of prototypes. In addition, ROSIE's English-like syntax can be used to produce highly readable code. With ROSIE, a knowledge engineer can extract rules of expertise approximately three times faster than with ROSIE's non-English-like counterparts.⁵ Such rules, coded in the terminology of the target domain, can later be used as formal documentation when transforming the prototype into a "mature" expert system.

1.6 CLOSING REMARKS

The ROSIE Language Development Project ran to completion in December of 1985. The final product of this effort is the version 3.0 release of ROSIE. ROSIE 3.0 is a complete reimplementations of ROSIE in PSL (Portable Standard Lisp) (Galway, 1984). We initiated the move to PSL as a means of improving ROSIE's performance and increasing its availability; PSL requires less memory, executes faster, and runs under a wider range of host environments than Interlisp (Teitelman, 1978), the language in which ROSIE was first developed. In the process of porting ROSIE to PSL, we took the opportunity to refine and enhance the design of the language.

Our primary goal in reimplementing ROSIE was not to change the basic definition of the language but to simplify and modularize its internal structure. This was done without a loss of generality or functionality. Thus, while making extensive changes to the major components of ROSIE, we have maintained a high degree of upward compatibility with existing code. Some of the outward changes, such as the introduction of filesegments as a data type, have been discussed (though not explicitly pointed out) earlier in this report. Other changes include the treatment of strings as two-dimensional ragged arrays and the inclusion of a new terse syntax for patterns. Overall performance improvements have been substantial, and certain components, such as the parser, file package, and string pattern matcher--which had previously caused significant bottlenecks in the programming process--run two to five times faster in ROSIE 3.0 than in any earlier release.

Although the development of ROSIE is no longer an active project at The RAND Corporation, the ROSIE environment is still being used as a testbed for research. One example of this is the experimental explanation facility being studied in *XPLROSIE* (Waterman et al., 1986). There is also work being done on melding a frame-based programming paradigm into the ROSIE environment. Finally, there is a group

⁵D. A. Waterman, personal communication.

examining how the knowledge acquisition process might be automated to the point of allowing knowledge engineers to develop large and robust ROSIE systems from a relatively small number of interviews with the domain expert. The ultimate goal of all this work is to overcome some of ROSIE's weaknesses and to provide knowledge engineers with a flexible environment in which expert rules can be formulated and later transformed into a high-performance expert system.

II. SYNTACTIC STRUCTURE

A fundamental task when learning how to program in ROSIE is learning the language's syntactic rules and how to apply them effectively. This is *not* a trivial task. While ROSIE appears to "understand" an impressive subset of the English language, it draws the sum total of its "knowledge" from a context-free grammar and actually has very little real understanding of English at all. Novices and experienced programmers alike often forget this point, writing very English-like code, which is completely incomprehensible to ROSIE.

Before ROSIE code can be executed, it must be translated or *parsed* into a machine-executable representation. Parsing consists of three phases: (1) *tokenization*, (2) *parse tree generation*, and (3) *transformation*. In this chapter, we will be discussing the first two phases of parsing; the third phase does not directly concern the user and will not be discussed.

Most of the design decisions regarding ROSIE's syntactic structure follow immediately from the choices made for its linguistic structures. Familiarity with ROSIE's linguistic structures (presented in succeeding chapters) may help novice users gain a better grasp of ROSIE's syntax. First-time readers are recommended to give the following sections only a cursory glance, returning to them after reviewing the rest of the manual.

2.1 PARSING BASICS

Parsing is the process by which a ROSIE source program is transformed into a machine-executable representation called *HILEV*. Parsing follows a three-step process: (1) lexical analysis (or *tokenization*); (2) *parse tree generation*; and (3) *transformation*. Tokenization is the process of breaking up words and special characters of the source code into *tokens* and grouping tokens together into independent *file items*. Parse tree generation is the process of deriving a unique parse tree for a given file item that represents its syntactic structure. Transformation is the process of mapping a parse tree into its HILEV representation.

To illustrate, consider parsing a file containing the ruleset,

To diagnose a situation:

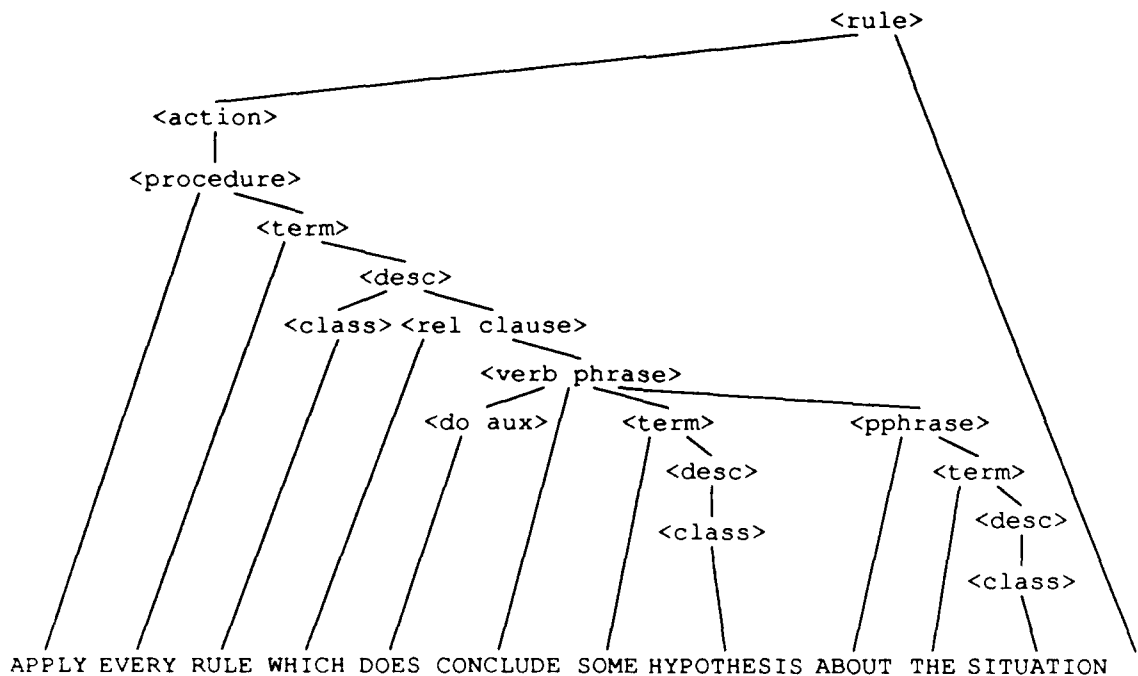
- [1] Apply every rule that does conclude some hypothesis about the situation.
- [2] For each rule that does conclude some true hypothesis about the situation,
print "{that rule} does apply, concluding:
 \ {that hypothesis}".

End.

Tokenization would decompose this into the four lists of tokens,

```
(TO DIAGNOSE A SITUATION :)
(APPLY EVERY RULE THAT DOES CONCLUDE SOME
 HYPOTHESIS ABOUT THE SITUATION .)
(FOR EACH RULE THAT DOES CONCLUDE SOME TRUE
 HYPOTHESIS ABOUT THE SITUATION , PRINT { {
 THAT RULE } " does apply, concluding" , CR ,
 " " , { THAT HYPOTHESIS } } .)
(END .)
```

representing four separate and syntactically independent file items. The second file item (starting APPLY EVERY RULE . . .) generates the parse tree,¹



¹Actually this is only an approximation; the real parse tree for this file item would be somewhat more complex.

for which the transformation process would produce the LISP expression,

```
(<RULE>
  (<DO>
    (<FOREVERY>
      (<DESC> <G0282> (<IDENT> RULE)
        (<IFSOME> (<DESC> <G0283> (<IDENT> HYPOTHESIS) NIL NIL)
          (<PROVABLE> 'DOESBIN 'PRESENT T
            (<USRVAR> <G0282> RULE)
            (<SYSVAR> <G0283>)
            (<IDENT> CONCLUDE
              (ABOUT (<THE> (<DESC> <G0284>
                (<IDENT> SITUATION) NIL NIL))))))
        (THAT DOES CONCLUDE SOME HYPOTHESIS ABOUT THE SITUATION))
      (<GO> (<IDENT> APPLY (*OBJECT* (<SYSVAR> <G0282>))))))
```

as its HILEV representation.²

2.2 TOKENIZATION

Lexical analysis of ROSIE source code is a process of recognizing *file items* and generating a list of representative *tokens* for each. This process is referred to as *tokenization*, and the lexical analyzer as the *tokenizer*. Below is an example of a ROSIE rule (a file item) and the list of tokens that would be produced for it.

Assert [that] <"item 3,4", 3.4 KG/M², Dave's age> is a tuple.

```
(ASSERT < "item 3,4" , 3.4 KG/M2 , DAVE ' S AGE > IS A TUPLE .)
```

One should observe that ROSIE is not case sensitive, except with respect to strings (e.g., "item 3,4"), and that tokenization returns all atomic tokens in uppercase. Notice also that comments--any characters surrounded by matching left and right square brackets ([])--are not part of the resulting list of tokens.

The tokenizer follows a simple algorithm. It is either *scanning* for a token, *collecting* the characters of a token, or *processing* those characters into a token. *Scanning* involves reading characters one at a time and throwing away those that do not start tokens, such as blanks, tabs, and new lines. *Collecting* means gathering characters that make up a token in a buffer. Finally, *processing* means converting characters from the buffer into either an *atomic*, *numeric*, or *string* token. Once a token has been processed, it is added to a list of tokens that correspond to a file item. This list is eventually returned as the result of tokenization.

²ROSIE users are never expected to deal with HILEV directly, therefore the definition of the various HILEV functions is not pertinent to this discussion.

2.2.1 The Character Set

In order to distinguish one token from the next, characters are divided into one of eight *character classes*. When the tokenizer reads a character, the character's class tells the tokenizer what to do next. For instance, it can either add the character to the token being built, finish the current token and start a new token, or terminate the tokenization process all together.

The eight character classes and the characters that belong to each are described below:

- *separator characters*: blanks, tabs, line feeds,
and all other control and
nonprinting characters

These characters delimit (or separate) tokens and are otherwise ignored. If the tokenizer encounters a separator character while scanning, it simply ignores the character and moves on to the next. If the tokenizer is in the collecting phase, it processes the contents of the token buffer and starts scanning for the next token.

- *break characters*: { } | ' ' () , < > ~ = ; :

These characters are recognized as single character tokens. When the tokenizer encounters a break character, it processes the contents of token buffer (if any), after which it adds a token representing the break character to the token list and begins scanning for a new token.

NOTE: The arithmetic operators (i.e., + - * / ^) are *not* break characters; thus, **3+4** constitutes a single token, while **3 + 4** constitutes three separate tokens.

- *string delimiter*: "

This character introduces and terminates strings. By default, it is the double quote ("), and all succeeding references to the double quote should be understood as references to the string delimiter character.

Normally, the characters encountered between a pair of matching double quotes become the characters of the resulting string. When the opening double quote is encountered, the contents of the token buffer (if any) are processed, after which the tokenizer starts collecting the characters of the string.

In ROSIE 3.0, the lexical syntax of strings has been substantially enhanced from previous versions of ROSIE. This new syntax will be discussed later in Section 2.2.4.

- *comment characters:* []

Square brackets delimit comments, which are otherwise ignored by the tokenizer with the exception that comments can be nested. When the open comment character ([) is encountered, the tokenizer scans past characters to the matching close comment character (]). The entire comment is treated as though it were a single separator character.

- *escape character:* \

This character tells the tokenizer to treat the next character as though it were of the class *letter*. The escape character is otherwise ignored and will not appear among the resulting tokens.

NOTE: The escape character is *not* recognized within comments and can take on special properties when found in a string.

- *terminator characters:* . ! ? :

These characters can (but do not always) indicate to the tokenizer that it has reached the end of a file item.

A terminator character is only recognized as such if it is the last nonseparator character on a line of text (where either an end-of-line character or end-of-file character can indicate the end of a line of text). An additional restriction maintains that the colon (:) can function as a terminator only when the first token of the item is one of **to**, **system**, or **before**, i.e., the colon is a terminator character only when it terminates a ruleset header.

If a character does not satisfy these restrictions, then it is not treated as a terminator. The characters (. ! ?) are treated as belonging to the class of *letters* and the colon (:) to the class of *break characters*.

- *digits:* 0-9

When encountered, a digit is added to the current token or starts a new one if none yet exists.

- *letters:* a-z A-Z - + / ^ * @ # \$ % & _

Most printing characters belong to this class. When encountered, a letter is added to the current token or starts a new one if none yet exists.

In the processing phase, if the contents of a token buffer can be interpreted as a numeric or string token, then that type of token is created. Otherwise, the characters are coerced into an atomic token.

2.2.2 Tokens and File Items

When the tokenizer is invoked, it successively reads characters from an input source, such as a file or the user's terminal, until it encounters one of the terminating characters (. ! ? :). Reaching such a character indicates to the tokenizer that it has recognized one *file item*. It returns a list of *tokens* created from intervening characters as the representation of that item. There are three types of tokens and five types of file items.

A token can be one of the following types:

- *numeric tokens*

which include integers of the form,

$[+/-]nnn$

and real numbers³ of the form,

$[+/-][nnn].nnn[E[+/-]nnn]$
 $[+/-]nnn.[nnn][E[+/-]nnn]$
 $[+/-]nnnE[+/-]nnn$

where *nnn* represents 1 or more digits, e.g.,

-23
 3.14
 .2
 2.
 -1.25E-9

Numeric tokens can optionally be preceded by a unary plus (+) or minus (-) and followed by an exponent.

- *string tokens*

which begin and end with a double quote and include all characters encountered between the double quotes. A double quote can be included in a string by *escaping* it, e.g.,

"MacArthur said, \"I shall return.\""

³If the number *nnn*. (e.g., *Let the counter be 1.*) appears as the last token on a line, then it will be recognized as an integer rather than a real, where the decimal point is treated as a terminator.

- *atomic tokens*

which covers any token not recognizable as a numeric or string token. Thus, the following

```
3.1.2
JK=DK
3+4
+4DESIGNS
OIL&GAS
```

are examples of unusual, but legal, atomic tokens.

A token's type is used when generating the parse tree representation of the file item. When a rule from ROSIE's BNF requires a particular type of token, it will be specified as *<atom>*, *<string>*, *<number>*, or *<integer>*, where *<integer>* will only match a numeric token that is a positive integer. These nonterminals will match any nonreserved token (see Section 2.3) of the designated type.

There are five types of file items recognized by the tokenizer:

- *ruleset headers*

which introduce ruleset definitions. A ruleset header can start with one of **to**, **system** or **before**, e.g.,

```
To move a ship to a destination:
System ruleset to move a ship to a destination:
Before executing to move a ship to a destination:
```

and is always terminated by a colon (:).

- *declarations and rules*

which include *private class declarations*, e.g.,

```
Private: a counter (initially 0), a reply.
```

execution monitor declarations, e.g.,

```
Execute cyclically.
```

ruleset rules and *file rules*, e.g.,

```
Assert battalion #5 was diverted to sector #7.
```

and *end statements*, e.g.,

```
End.
```

These file items appear as a sequence of tokens terminated by a period (.).

- *queries*

which are special command forms recognized by the top-level monitor and within a break loop, e.g.,

```
??  
Conclusions?  
14?
```

They are characterized as terminating with a question mark (?).

- *break commands*

which are special command forms recognized only within a break loop, e.g.,

```
Eval!  
Resume ruleset!
```

They are characterized as terminating with an exclamation point (!).

2.2.3 Comments

Comments can appear anywhere in ROSIE source code. Comments are preceded by a left square bracket ([) and terminated by a right square bracket (]). Additionally, comments can be nested to any depth.

When a comment appears within the text of a file item, it is treated like a separator character. This means that if the tokenizer were in the collecting phase, it would process the token, add it to the list of tokens, and start scanning for a new token. Comments that appear outside of a file item are likewise ignored. The text of such comments will be associated with the text of the file item immediately following them, meaning that when a file item is edited or examined via the file package, the comments immediately preceding it will appear with the text of the file item as well.

There are several restrictions upon comments appearing outside of a file item:

- 1) If a comment precedes a file item, but begins on the same line as that item, e.g.,

```
[2] Assert . . .
```

its text will be discarded by tokenizer.

- 2) If a comment follows a file item, but begins on the same line as that item, e.g.,

End. [End of report info]

it will be included with the text of that file item, rather than the text of the next file item.

- 3) If a comment (or sequence of comments) is not otherwise followed by a file item, it is discarded.

ROSIE automatically generates comments to attach rule numbers to file rules and ruleset rules. It supplies the numbers and keeps them up-to-date whenever a program file is edited or otherwise modified.

It has been our experience that ROSIE programs require far fewer comments than is typical of other programming languages. Despite the fact that claims to a "self-documenting" language have been made in the past, ROSIE code is remarkably expressive and needs little documentation.

2.2.4 Extended String Syntax

Among the changes to appear in ROSIE 3.0 is an extension to the lexical syntax of strings. This syntax allows the otherwise verbose and unwieldy syntax of *pattern elements* (see Section 9.6) to be captured in a form that outwardly appears to be a single string token, thus enhancing both the clarity and readability of patterns. The new syntax subsumes the older form (i.e., zero or more characters delimited by double quotes) and in most cases is compatible with strings in existing code.

When the tokenizer encounters a double quote ("), it begins reading a string. If a matching double quote is encountered before the end-of-line character, the tokenizer creates a string token of the form "ccc", where *ccc* are the characters appearing between the opening and closing quotes. Thus, when reading the characters

"This is a string"

the tokenizer will treat them as constituting a string token.

However, if the end-of-line character is encountered before the closing double quote, the tokenizer assumes an instance of the extended string syntax, the result of which will be a list of tokens specifying a pattern. For example, scanning the characters

**"East is east and west is west,
and never the twain shall meet"**

generates the tokens

```
{ "East is east and west is west" , CR ,
  "and never the twain shall meet" }
```

These tokens would be parsed as a pattern; the pattern can be coerced back into a *string element* (see Section 9.5). Terms and subpatterns may be embedded in strings by delimiting them with matching left and right curly braces ({ }).

To be precise, the lexical analysis of strings is guided by the following rules:

- Strings are delimited by a pair of matching double quotes ("");
- Characters appearing in a string and surrounded by a pair of left and right curly braces ({ }) are scanned as though they existed outside of the string, e.g.,

```
"Is {the hypothesis} correct? "
```

will be scanned as the pattern

```
{ "Is " , { THE HYPOTHESIS } , " correct? " }
```

Note also that

```
"This {"is a trick"} string"
```

will be scanned as

```
{ "This " , { "is a trick" } , " string" }
```

- When the scanner reaches an end-of-line character before a closing double quote, it inserts a carriage return subpattern (CR) into the resulting list of tokens, separating the characters on one line from the characters on the other, e.g.,

```
"foo
fum"
```

will be scanned as

```
{ "foo" , CR , "fum" }
```

The string to the left of the carriage return will end with the last nonseparator character encountered before the end-of-line character. The string to the right of the carriage return will start with the first nonseparator character encountered after the end-of-line character; all intervening separator characters (with the exception of the end-of-line character that always inserts an

additional carriage return) are ignored.

As an example, consider the string

```
"Airfield: {the airfield} Target: {the target}
  Capabilities are {the capabilities of that target}
  Vulnerability is {the vulnerability of that target}"
```

which will be scanned as

```
{ "Airfield: " , { THE AIRFIELD } , " Target: " ,
{ THE TARGET } , CR , "Capabilities are " , { THE
CAPABILITIES OF THAT TARGET } , CR , "Vulnerability is "
, { THE VULNERABILITY OF THAT TARGET } }
```

Note that the indentations of the source string are not preserved.

- If a back slash (\) is the first or last character of a line, then it (and its use as the escape character) is ignored. All separator characters that respectively follow or precede the back slash will appear in the resulting tokens, e.g.,

```
"Airfield: {the airfield} Target: {the target}
\ Capabilities are {the capabilities of that target}
\ Vulnerability is {the vulnerability of that target}"
```

will be scanned as

```
{ "Airfield: " , { THE AIRFIELD } , " Target: " ,
{ THE TARGET } , CR , " Capabilities are " ,
{ THE CAPABILITIES OF THAT TARGET } , CR ,
" Vulnerability is " , { THE VULNERABILITY OF THAT
TARGET } }
```

This provides a simple technique for preserving indentation.

- The special attributes of any character appearing in the string will be ignored if immediately preceded by the escape character, e.g.,

```
"MacArthur said, \"I shall return.\""
```

will be scanned as the token,

```
"MacArthur said, "I shall return.""
```

Note that this use of the back slash is superseded when the back slash is the first or last character on a line.

- No other characters or character strings have any special

significance unless they appear between left and right curly braces.

- If none of the above features (with the exception of the escape character) are encountered while scanning a string, then the string will be returned as a single string token. Otherwise, it will be returned as a list of tokens designating a pattern element.

2.3 RESERVED WORDS

Reserved words are tokens that can only appear as literals in specific syntactic constructs. All break characters are reserved words, as are a large set of prepositions, and the auxiliary forms of *be* and *do*. The following is a list of the reserved words used in ROSIE 3.0:

(1) { } | ' ' () , < > ~ = ; : . ? !

(2) + - * / ^ **

(3)	ABOUT	AT	DURING	ON	THRU	WITHIN
	ABOVE	BECAUSE	FOR	ONTO	TO	WITHOUT
	ACROSS	BEFORE	FROM	OUT	TOWARD	
	AFTER	BEHIND	IN	OUTSIDE	UNDER	
	AGAINST	BELOW	INSIDE	OVER	UNTIL	
	ALONG	BESIDE	INTO	PER	UP	
	AMONG	BETWEEN	NEAR	SINCE	VIA	
	AROUND	BY	OF	THAN	WHILE	
	AS	DOWN	OFF	THROUGH	WITH	

(4)	AM	DID	DOES	WAS	WERE	WILL
	ARE	DO	IS			

(5)	A	ANY	EITHER	SOME	THE	THAT
	AN	EACH	EVERY	SUCH		

(6)	WHERE	WHICH	WHO	WHOM	WHOSE	EXCEPT
-----	-------	-------	-----	------	-------	--------

(7)	AND	HAS	LET	THERE	UNLESS	OTHERWISE
	ELSE	IF	NOT			

These are arranged (loosely) according to usage, i.e., (1) *break characters*, (2) *arithmetic operators*, (3) *prepositions*, (4) *auxiliary forms of be and do*, (5) *noun phrase specifiers*, (6) *relative clause specifiers*, and (7) *other reserved words*.

2.4 THE ROSIE GRAMMAR

The Backus-Naur form (BNF) (Pagan, 1981) description of ROSIE's syntax is provided below for users who want a terse but complete definition of the language. The following conventions apply:

- Uppercase indicates terminal symbols (i.e., tokens that must appear in a sentential segment in order for that segment to be recognized as an instance of a grammar rule);
- Angle brackets (< >) surround nonterminal symbols (i.e., symbols that are found in the left-hand side of a production);
- There are four "special" nonterminal symbols, which match a single token according to type:

`<string>` matches any string token;
`<number>` matches any numeric token;
`<integer>` matches any positive integer;
`<atom>` matches any nonreserved atomic token.

A simple lexical BNF for these token types is provided below;

- Square brackets ([]) surround optional constructs and appear when the ordering of alternative productions of a nonterminal is unimportant;
- Vertical bar (|) separates alternate constructs;
- The epsilon symbol (ϵ) denotes null productions.

2.4.1 The Lexical BNF

The lexical BNF describes (in a limited fashion) the syntax of tokens. In a ROSIE source program, tokens are normally delimited by separator and break characters. In the following rules, *nnn* stands for any sequence of one or more digits, and *ccc* stands for any sequence of letters or digits.

```
<token> ::= <number>
          ::= <string>
          ::= <atom>

<number> ::= <integer>
          ::= <real>

<integer> ::= [+|-]nnn
```

```

<real>      ::= [+|-]nnn.[nnn][E[+|-]nnn]
             ::= [+|-][nnn].nnn[E[+|-]nnn]
             ::= [+|-]nnnE[+|-]nnn

```

```

<string> ::= ""
           ::= "ccc"

```

```

<atom> ::= ccc

```

2.4.2 The Linguistic BNF

Once the tokenizer has produced a set of file items representing a source program, control passes to the *driver routines*. The driver attempts to identify each file item as an instance of the nonterminal symbol <program>, which represents the start symbol of the BNF seen below. Note that rulesets are not a part of this BNF; rulesets are organized from their components after parsing.

```

<program> ::= <declaration>
           ::= <query cmd>
           ::= <rule>

<query cmd> ::= ? ?
             ::= <integer> ?
             ::= <name element> ?
             ::= ?

```

NOTE: <name element> is a sequence of one or more <atom>.

```

<declaration> ::= EXECUTE <monitor> .
               ::= PRIVATE <class list> .
               ::= PRIVATE : <class list> .
               ::= END .
               ::= <header> :
               ::= SYSTEM RULESET <header> :

<class list> ::= <formal> [( [INITIALLY] <term> )] [, <class list>]

<monitor> ::= SEQUENTIALLY
            ::= RANDOMLY
            ::= CYCLICALLY

<header> ::= TO GENERATE <genr form>
          ::= BEFORE GENERATING <genr form>
          ::= BEFORE PRODUCING <genr form>

          ::= TO DECIDE [IF] <pred form>
          ::= BEFORE TESTING [IF] <pred form>

          ::= TO <proc form>
          ::= BEFORE INVOKING <proc form>

```

::= BEFORE ASSERTING <pred form>
 ::= BEFORE DENYING <pred form>

<genr form> ::= [<determiner>] <root name> [<private pps>]

<proc form> ::= <atom> [<formal>] [<private pps>]

<pred form> ::= <formal> <be aux> <a/an> <root name> [<private pps>]
 ::= <formal> <be aux> <atom> [<formal>] [<private pps>]
 ::= <formal> <be aux> <prep> [<formal>] [<private pps>]
 ::= <formal> <do aux> <atom> [<formal>] [<private pps>]

<determiner> ::= THE | <a/an>

<a/an> ::= A | AN

<root name> ::= <atom list>

<private pps> ::= <prep> <formal> [<private pps>]

<formal> ::= [<a/an>] <root name>

<opt pphrase> ::= <pphrase>
 ::= ε

<pphrase> ::= <pp> <pphrase>
 ::= <pp>

<pp> ::= <prep> <term>
 ::= (<prep> <term>)

NOTE: It is important to notice the order of productions for prepositional phrases. This ordering specifies that the longest chain of prepositions is always preferred. Compare the use of prepositions in the <procedure> and <proposition> rules to that in the <description> rules. The fundamental difference is that the former use the <opt pphrase> production while the latter uses the <pphrase> production.

<rule> ::= <action block> .

<action block> ::= <action>
 ::= <action> AND <action block>

NOTE: The highest level <action block> will take the shortest chain of <action>.

<action> ::= (<action block>)
 ::= <data action>
 ::= <iter action>
 ::= <cond action>

```

::= <cond block>
::= <exec action>
::= <procedure>

<data action> ::= ASSERT <prop block>
               ::= DENY <prop block>
               ::= LET <let block>
               ::= CREATE <a/an> <description>

<prop block> ::= <proposition>
               ::= <proposition> AND <prop block>

<let block> ::= <let form>
               ::= <let form> AND <let block>

<let form> ::= THE <description> BE <term>
             ::= <term> ' S <description> BE <term>
             ::= <term> BE THE <description>
             ::= <term> BE <term> ' S <description>

<cond action> ::= IF <condition> <then part>
               ::= IF <condition> <then part> <else part>
               ::= UNLESS <condition> <then part>
               ::= UNLESS <condition> <then part> <else part>

```

NOTE: The <else part> attaches to the deepest <cond action>.

```

<then part> ::= , [THEN] <action block> [,]
             ::= THEN <action>
             ::= ( <action block> )

<else part> ::= OTHERWISE , <action block> [,]
             ::= OTHERWISE <action>
             ::= ELSE , <action block> [,]
             ::= ELSE <action>

<cond block> ::= <select block>
               ::= <choose block>
               ::= <match block>

<select block> ::= SELECT <term> : <select form> [;] <default block>

<select form> ::= <tuple element> <action block>
               ::= <tuple element> <action block> ; <select form>

<match block> ::= MATCH <term> : <match form> [;] <default block>

<match form> ::= <pattern element> <action block>
               ::= <pattern element> <action block> ; <match form>

<choose block> ::= CHOOSE SITUATION : <choose form> <default block>

```

```

<choose form> ::= IF <condition> <then part>
               ::= IF <condition> <then part> ; <choose form>

<default block> ::= ε
                 ::= DEFAULT : <action block> [;]

```

NOTE: The <action block> and <default block> always attach to the most deeply embedded conditional block.

```

<iter action> ::= <iter block> , <action block> [,]
               ::= <iter block> ( <action block> )

<iter block>  ::= <for part> [<while part>] [<until part>]
               ::= <while part> [<until part>]
               ::= <until part>

<for part>   ::= FOR EACH <description>

<while part> ::= WHILE <condition>

<until part> ::= UNTIL <condition>

<procedure> ::= DO NOTHING
              ::= PRODUCE <term>
              ::= DISPLAY <term>
              ::= <atom> [<term>] <opt pphrase>

```

NOTE: The **do nothing**, **produce**, and **display** procedures are all defined specially; **do nothing**, because **do** is a reserved word and will not match <atom>; **produce** and **display**, because their normal usage does not conform to the default syntax.

```

<condition> ::= <disjunct>
              ::= <comma or>
              ::= <comma and>

<comma or>  ::= <disjunct> [, OR <comma or>]

<comma and> ::= <disjunct> [, AND <comma and>]

```

NOTE: Sequences of ', OR' and ', AND' may not appear in the same <condition> unless delimited with parentheses.

```

<disjunct> ::= <conjunct> OR <disjunct>
            ::= <conjunct>

<conjunct> ::= <primary> AND <conjunct>
            ::= <primary>

<primary>  ::= ( <condition> )

```


::= <sentence>

NOTE: AND has precedence over OR; precedence can be overridden with parentheses.

<sentence> ::= <proposition>
 ::= <special form>

<proposition> ::= <term> <verb phrase>

<verb phrase> ::= <be aux> <a/an> <description>
 ::= <be aux> <atom> [<term>] <opt pphrase>
 ::= <be aux> <prep> [<term>] <opt pphrase>
 ::= <do aux> <atom> [<term>] <opt pphrase>

<be aux> ::= WAS [NOT]
 ::= WERE [NOT]
 ::= AM [NOT]
 ::= ARE [NOT]
 ::= IS [NOT]
 ::= WILL [NOT] BE

<do aux> ::= DID [NOT]
 ::= DO [NOT]
 ::= DOES [NOT]
 ::= WILL [NOT]

<special form> ::= THERE IS <how many> <description>
 ::= THERE IS SUCH <a/an> <class noun>
 ::= <term> <special vp> [(<desc var>)]

<how many> ::= NO
 ::= <a/an>
 ::= JUST ONE
 ::= MORE THAN ONE

<special vp> ::= HAS <how many> <description>
 ::= HAS SUCH <a/an> <class noun> [(<desc var>)]
 ::= <rel op> <term>

<rel op> ::= IS [NOT] EQUAL TO
 ::= IS [NOT] GREATER THAN [OR EQUAL TO]
 ::= IS [NOT] LESS THAN [OR EQUAL TO]

::= =
 ::= ~=
 ::= >
 ::= ~>
 ::= >=
 ::= ~>=
 ::= <

```

::= ~<
::= <=
::= ~<=

```

```

<term> ::= <subterm>
        ::= <arith expr>

```

```

<arith expr> ::= <arith expr> + <mult expr>
               ::= <arith expr> - <mult expr>
               ::= <mult expr>

```

```

<mult expr> ::= <mult expr> * <expt expr>
               ::= <mult expr> / <expt expr>
               ::= <expt expr>

```

```

<expt expr> ::= <subterm> ^ <expt expr>
               ::= <subterm> ** <expt expr>
               ::= <subterm>

```

NOTE: Precedence Associativity

()	
^ **	right
* /	left
+ -	left

```

<subterm> ::= ( <term> )
           ::= <iter term>
           ::= <desc term>
           ::= <anaphora>
           ::= <element>

```

```

<iter term> ::= ONE OF <term list> [,] OR <term>
              ::= EITHER <term list> [,] OR <term>
              ::= EACH OF <term list> [,] AND <term>
              ::= BOTH <term> AND <term>

```

```

<term list> ::= <term> [, <term list>]

```

```

<desc term> ::= THE TUPLE CONTAINING EACH <description>
              ::= <term> ' S <description>
              ::= THE <description>
              ::= A NEW <description>
              ::= <a/an> <description>
              ::= SUCH A NEW <class noun> [( <desc var> )]
              ::= SUCH <a/an> <class noun> [( <desc var> )]
              ::= SOME <description>
              ::= EVERY <description>

```

NOTE: <term>'S <description> is equivalent to <description> OF <term>

```

<description> ::= SUCH <class noun> [( <desc var> )]

```

```

::= <class>
::= <class> <rel clause>

```

NOTE: A <rel clause> will always attach to the rightmost <description>.

```

<class> ::= [<root name>] <class noun> [( <desc var> )]
        ::= [<root name>] <class noun> [( <desc var> )] <pphrase>

```

NOTE: A <class> will try to take no prepositional attachments. Failing that, the leftmost <class> will attempt to take the longest chain of prepositions.

```

<root name> ::= <atom> [<root name>]

<class noun> ::= <atom>

<desc var> ::= <atom>

<rel clause> ::= <disj clause>

<disj clause> ::= <conj clause> [OR <disj clause>]

<conj clause> ::= <clause form> [AND <conj clause>]

```

NOTE: AND has precedence over OR; precedence can be overridden with parentheses.

```

<clause form> ::= ( <rel clause> )
               ::= <such that/where>
               ::= <that/which/who>
               ::= <whose>
               ::= <which/whom>
               ::= <except>

<such that/where> ::= ( <st/w> <condition> )
                   ::= <st/w> <primary>

<st/w> ::= SUCH THAT
        ::= WHERE

<that/which/who> ::= <t/w/w> [<term>] <verb phrase>
                  ::= <t/w/w> <special vp>
                  ::= <t/w/w> <term> <rel op>

<t/w/w> ::= THAT
        ::= WHICH
        ::= WHO

<whose> ::= WHOSE <description> <be aux> <term>

```

```

<which/whom> ::= <prep> <w/w> <term> <verb phrase>

<w/w> ::= WHICH
        ::= WHOM

<except> ::= EXCEPT <term>

<anaphora> ::= THAT <class noun>
            ::= <rule var>

<rule var> ::= <desc var>

<element> ::= <name element>
            ::= <number element>
            ::= <tuple element>
            ::= <string element>
            ::= <pattern element>
            ::= <filesegment>
            ::= <class element>
            ::= <intentional description>
            ::= <intentional proposition>
            ::= <intentional procedure>

<name element> ::= <atom list>

<atom list> ::= <atom> [<atom list>]

<number element> ::= <number>
                  ::= <number> <atom list>
                  ::= <atom list> <number>

<tuple element> ::= < [ <term list> ] >

<string element> ::= <string>

<pattern element> ::= { <pat spec> }

<filesegment> ::= ' FILE : <string> [, <header>] [, <rule spec>] '
                ::= ' <header> [, <rule spec> ] '

<rule spec> ::= BEFORE <term>
              ::= AT <term>
              ::= FROM <term> TO <term>
              ::= AFTER <term>
              ::= <integer>
              ::= <integer> <integer>

<class element> ::= ANY <description>

<intentional description> ::= ' THE <description> '
                          ::= ' <term> ' S <description> '

```

```

::= ' <a/an> <description> '
::= ' SUCH <a/an> <class noun> '

```

```

<intentional proposition> ::= ' <proposition> '

```

```

<intentional procedure> ::= ' <procedure> '

```

```

<pat spec> ::= FIXED FORMAT <pat conj>
           ::= FREE FORMAT <pat conj>
           ::= ADJOIN <pat conj>
           ::= <pat disj>

```

```

<pat disj> ::= <pat conj> | <pat disj>
           ::= <pat conj>

```

```

<pat conj> ::= <bind spec> , <pat conj>
           ::= <bind spec>

```

NOTE: The conjunctive (,) has precedence over the disjunctive (|); precedence can be overridden with curly braces.

```

<bind spec> ::= <rep spec>
           ::= <rep spec> ( BIND TO <bind form> AS <bind type> )
           ::= <rep spec> ( BIND TO <bind form> )
           ::= <rep spec> ( BIND <bind form> TO <bind type> )
           ::= <rep spec> ( BIND <bind form> )
           ::= <rep spec> ( BOUND TO <bind form> )
           ::= BIND <rep spec> TO <bind form> AS <bind type>
           ::= BIND <rep spec> TO <bind form>

```

```

<bind type> ::= <a/an> NAME
            ::= <a/an> NUMBER
            ::= <a/an> STRING
            ::= <a/an> TUPLE
            ::= <a/an> PATTERN
            ::= <a/an> CLASS
            ::= <a/an> DESCRIPTION
            ::= <a/an> PROPOSITION
            ::= <a/an> PROCEDURE
            ::= <a/an> FILESEGMENT
            ::= <a/an> ELEMENT

```

```

<bind form> ::= <desc var>
            ::= THE <description>
            ::= <term> ' S <description>

```

```

<rep spec> ::= [<rep form> [OF]] <subpat>
           ::= ANYTHING
           ::= SOMETHING

```

```

<rep form> ::= <integer>

```

```

::= <integer> OR MORE
::= <integer> OR LESS
::= <integer> OR FEWER

```

```

<subpat> ::= { <pat spec> }
::= BOX <subpat> TO WIDTH <term>
::= PAD <subpat>
::= LEFT JUSTIFY <subpat> [<box size>]
::= LJ [<term> [BY <term>]] : <subpat>
::= RIGHT JUSTIFY <subpat> [<box size>]
::= RJ [<term> [BY <term>]] : <subpat>
::= CENTER JUSTIFY <subpat> [<box size>]
::= CJ [<term> [BY <term>]] : <subpat>
::= CODES ( <int list> )
::= BELL[S]
::= TAB[S]
::= EOL[S]
::= BACKSPACE[S]
::= BS
::= PAGE[S]
::= FORMFEED[S]
::= ESCAPE[S]
::= EOL[S]
::= END
::= BLANK[S]
::= QUOTE[S]
::= RETURN[S]
::= CR[S]
::= CHARCODE <term>
::= CONTROL <term>
::= LINE[S]
::= <char class> [[NOT] IN <term>]
::= <term>

```

```

<box size> ::= TO LENGTH <term> [AND WIDTH <term>]
::= TO WIDTH <term>

```

```

<coords> ::= AT < <term> , <term> >

```

```

<padding> ::= STARTING LEFT
::= STARTING RIGHT
::= CENTERING

```

```

<int list> ::= <integer> [, <int list>]

```

```

<char class> ::= [NON]ALPHANUMERIC[S]
::= [NON]BLANK[S]
::= [NON]CONTROL[S]
::= [NON]DIGIT[S]
::= [NON]LETTER[S]
::= [NON]NUMBER[S]

```

::= [NON]NUMERAL[S]
::= CHARACTER[S]

2.5 PARSE TREE GENERATION

Once the tokenizer has processed the characters of a file item, the resulting list of tokens is passed to the *parse tree generator* (PTG), which produces a tree representation of some rightmost derivation of the file item. The PTG is the most complex component of the parser.

The PTG, based upon fast multitrack parsing techniques for general context-free languages (Irons, 1971; Quinlan, unpublished working notes), resembles an LALR parser that follows all derivations of a file item in a breadth-first manner.⁴ It is responsible for detecting syntax errors as well as resolving ambiguities.

One feature of the PTG found in ROSIE 3.0 (but not found in earlier ROSIEs) is a complete and consistent set of disambiguation rules. These rules have eliminated the occurrence of ambiguity errors, even in the presence of prepositional phrases (a common source of such parsing errors in earlier ROSIEs). This feature enhances code readability by reducing the number of delimiting parentheses otherwise required to avoid surface-level ambiguities. Unfortunately, this feature is not without cost, adding an extra burden on the ROSIE programmer to ensure that his code is interpreted correctly.

2.5.1 Associativity, Precedence, and Disambiguation

The PTG derives its rules of precedence and associativity, and ultimately its rules for resolving ambiguities, out of the context-free grammar from which its parse table was compiled. For ROSIE 3.0, this is the grammar seen in Section 2.4. The rules of precedence and associativity describe how otherwise ambiguous sentence fragments will be interpreted.

In ROSIE 3.0, precedence and associativity are immediately decidable from the order of productions in the grammar. When a nonterminal can be derived from several alternative productions, the production appearing earliest in the grammar is preferred to the productions appearing later. Given two alternate derivations, the preferred derivation is selected via a depth-first, left-to-right comparison of the productions used in both. First, the productions at the roots of each parse tree are compared. If the same production appears in both positions, then the productions of the leftmost children are compared and so on. When the two productions being compared are different, then the parse tree using the preferred production is returned as the preferred derivation.

⁴This technique is similar to that found in Tomita (1985).

While the grammar in Section 2.4 should be used as the ultimate authority on questions of associativity, precedence, and disambiguation, we will not be so fiendish as to say that is all there is to know. Succeeding chapters covering linguistic structures with particularly problematic syntax (e.g., embedded actions and action blocks) will include discussions on the associativity and precedence of such structures. In addition, the file package provides a tool for "deparsing" file items--deparsing translates the HILEV representation of a file item into ROSIE source code, demonstrating the interpretation of that file item visibly through the use of indentation and parentheses.

2.5.2 The Disambiguation of Prepositional Phrases

The singularly most problematic of all ROSIE's syntactic forms is the *prepositional phrase*. Prepositional phrases are modifiers, attaching additional arguments to certain linguistic structures. They are allowed to modify the action verb of a *procedure*, e.g.,

move a ship from Le Havre to Auckland

the relational verb of a *proposition*, e.g.,

the ship did move from Le Havre to Auckland

and the *class noun* of a *description*,

the ship in dry-dock

Descriptions can further be modified by a *relative clause*, e.g.,

the ship in dry-dock which did move from Auckland

the main verb of which may likewise be modified by a prepositional phrase. Since *descriptive terms*, such as

the ship in dry-dock . . .

are commonly used as arguments to procedures and propositions as well as other descriptions, there is considerable room for ambiguity.

In places where an ambiguous parse is possible, the following two rules apply to the interpretation of prepositional phrases:

- A prepositional phrase will always modify the closest verb, if syntactically possible.
- If the surrounding context does not support a verb, a prepositional phrase will modify the class noun of the leftmost description preceding the preposition.⁵

⁵If the surrounding context does not support such a description, a syntax error occurs.

As an aside, a relative clause will attach itself to the description immediately preceding it; once attached, the class noun of that description cannot be further modified by a prepositional phrase.

To illustrate these rules, consider the following pairs of examples selected because their default interpretation is somewhat counterintuitive.

report the name of the man

report (the name) (of the man)

move the ship which does come from the port to the harbor

move (the ship (which does come (from the port) (to the harbor)))

the ship from the port on the coast did move to the harbor

(the ship (from the port) (on the coast)) did move (to the harbor)

the man did hit the lady from the city who is wearing blue

(the man) did hit (the lady) (from the city who is wearing blue)

Parentheses in the second statement of each example pair (regular font) delimit the extent of descriptive terms and relative clauses, demonstrating the default associations of prepositional phrases and relative clauses.

2.5.3 Directing Disambiguation with Parentheses

Users can direct the attachment of prepositional phrases (as well as other syntactic forms) with parentheses. Essentially, parentheses narrow the context of interpretation. With parentheses a user can indicate precisely the syntactic groupings desired.

As an example, consider the earlier unparenthesized examples. The intuitive interpretation of these statements will be made by ROSIE if we add parentheses as indicated below:

report (the name of the man)

move (the ship which does come from the port) to the harbor

the ship (from the port on the coast) did move to the harbor

the man did hit the lady (from the city) who is wearing blue

By surrounding **the name of the man** with parentheses, **report** is no longer in the context of interpretation, meaning the phrase introduced by **of** can only modify **name**. Alternatively, the change to the second example hides the verb **come** and allows the phrase introduced by **to** to correctly modify the verb **move**. The third example is similar to the

first; the fourth, however, is unique. Since the relative clause must attach to a description immediately preceding it, and it can no longer attach to **city**, the phrase introduced by **from** must attach to **lady** so that the relative clause can modify the description **lady from the city**.

III. RUNNING ROSIE

This chapter describes the general operating procedures for running ROSIE. While it is intended as a beginner's guide, those readers already familiar with ROSIE may find this information instructional as well. In this chapter, we present ROSIE's interactive working environment and discuss how one normally develops a program in this environment.¹

3.1 GETTING STARTED

ROSIE is an interactive programming system, and so in one respect is similar to LISP. To get started, enter a ROSIE session. In UNIX,² this can be done with the command `rosie`, which puts you into ROSIE's *top-level monitor*, e.g.,

```
% rosie
(R)
[ ROSIE Version 3.0 (PSL) 15-Apr-86 ]

<1>
```

During initialization of a ROSIE session, the file `.rosierc`,³ if present, will be loaded from the user's home directory. This file may only contain LISP expressions, which are evaluated at load time. Typically, these are commands to load specific user modules, set system switches (discussed in Appendix C), etc.

Upon entering the top level, ROSIE prints a banner line describing the version and date of your system, and then prompts you for the first line of input. From this point on, you are talking with ROSIE. All aspects of system development, testing, refinement, and maintenance can be done from the top-level monitor.

¹This chapter assumes ROSIE 3.0 running in PSL under UNIX 4.2. See your site consultant for system operating procedures specific to your installation.

²Assuming your paths are initialized to include the ROSIE/bin directory.

³This file name may be different for sites not running ROSIE under UNIX. Check with your site consultant.

3.2 INTERACTIONS AT THE TOP LEVEL

The top-level monitor is an interactive control loop that prompts the user with the current line number surrounded by angle brackets. The user commands ROSIE by issuing a *monitor rule* to the top level. The top level immediately parses and executes each monitor rule as it is received and then prompts for another.

ROSIE provides a *history facility* for interactions at the top level. This facility keeps track of the last 40 monitor rules issued. These can be examined by the user, re-executed, and even edited. ROSIE also provides a small set of monitor commands with an abbreviated syntax for examining the history list and the database structure.

The following sample session is provided to give the new user a feeling for working in ROSIE's top-level monitor. This demonstration is by no means complete, but it should make the reference manual easier to understand and your first attempt at interacting with ROSIE more successful.

```
(R)
[ ROSIE      Version 3.0 (PSL) 30-May-86 ]

<2> Assert John is a man.
<3> Assert each of Mary and Sara is a woman.
<4> ?
[ GLOBAL Database ]
    SARA IS A WOMAN.
    MARY IS A WOMAN.
    JOHN IS A MAN.
```

Programming in ROSIE consists primarily of actions performed on relations in the database. The database is built up by asserting propositions. Lines <2> and <3> above are examples of such assertions. In line <2>, we assert that the element **John** is a member of the class **man**. In line <3>, we make a similar assertion, which is applied to the elements **Mary** and **Sara** successively. In line <4>, we use the ? monitor command to examine the contents of the database.

Note: Line <1> is missing because it commanded ROSIE to **dribble** this session to a text file.

```
<5> Assert any man does like any woman.
<6> ?
[ GLOBAL Database ]
    ANY MAN DOES LIKE ANY WOMAN.
    SARA IS A WOMAN.
    MARY IS A WOMAN.
    JOHN IS A MAN.
```

The elements seen in lines <2> and <3> (i.e., **John**, **Mary**, and **Sara**) are called *names*. Line <5> gives an example of another type of element called *class elements*. Here, **any man** and **any woman** are both class elements. A class element implicitly represents any element that is a member of that class. Thus, **any man** represents any element, such as **John**, that satisfies the '*element is a man*' relation.

Another thing to notice is the use of **does** in line <5>. The main verb of any ROSIE proposition *must* be introduced by an auxiliary form of *be* or *do*. Hence, we have **does like** rather than **likes**.

```
<7> Display every woman that John does like.
SARA
MARY
```

In line <7> we have an example of a *quantified descriptive term*, i.e., **every woman that John does like**. This is distinguished from a class element by the use of the function word **every** rather than **any**. Instead of being an implicit placeholder for elements of the class described, this type of term explicitly evaluates to a sequence of those elements, each of which is passed to the **display** procedure in turn.

To find the elements of **woman that John does like**, ROSIE goes through the following process. First, it finds the elements of the class **woman**, and then it uses the relative clause as a filter on those elements, i.e., each element generated must satisfy the relation

John does like element

To decide if '**John does like Sara**', ROSIE examines the database for instances of the **does like** relation, comparing the target proposition to each until it can be confirmed or disproved. Comparison to the proposition asserted in line <5> confirms the proposition because **John** is a member of the class **man** and **Sara** is a member of the class **woman**. Thus, **Sara** can be passed to **display**. Likewise, **Mary** will go through the same tests and be passed to **display**.

```
<8> ??

<8> ??
<7> DISPLAY EVERY WOMAN THAT JOHN DOES LIKE.
<6> ?
<5> ASSERT ANY MAN DOES LIKE ANY WOMAN.
<4> ?
<3> ASSERT EACH OF MARY AND SARA IS A WOMAN.
<2> ASSERT JOHN IS A MAN.
<1> DRIBBLE TO "demo".

<9> Redo 7.
SARA
MARY
```

Lines <8> and <9> demonstrate the use of the history facility. The ?? monitor command displays up to the last 40 monitor rules issued, while the **redo** procedure allows the user to re-execute any of those rules.

```
<10> Deny any man does like any woman.
<11> ?
[ GLOBAL Database ]
  SARA IS A WOMAN.
  MARY IS A WOMAN.
  JOHN IS A MAN.
```

In line <10> we have an example of how to remove an assertion from the database. Note here that the denial of a proposition that contains a class element requires that the class element be used in the denial, i.e., we could not have used '**John does like any woman**'.

```
<12> Assert any man does like a woman.
<13> ?
[ GLOBAL Database ]
  ANY MAN DOES LIKE SARA.
  SARA IS A WOMAN.
  MARY IS A WOMAN.
  JOHN IS A MAN.
```

In line <12> we see an important difference between **any** and **a**. The article **a** introduces *simple descriptive terms*, and so is like **every** in the respect that it is an explicit rather than implicit reference to a member of a class, evaluating to the first element that can be generated from that class (i.e., **Sara**). This element then appears as an argument to the asserted proposition.

```
<14> Forget about every woman.
<15> ?
[ GLOBAL Database ]
  JOHN IS A MAN.
```

Here's another example of how to delete propositions from the database. The **forget about** procedure causes every proposition containing an instance of its argument to be removed from the database. As seen before in line <7>, the quantifier **every** causes this procedure to be applied to each member of the class **woman**.

```
<16> Assert any man does like a woman.
<17> ?
[ GLOBAL Database ]
  ANY MAN DOES LIKE WOMAN #1.
  WOMAN #1 IS A WOMAN.
  JOHN IS A MAN.
```

In this example, we find another property about the article **a**. When there is no element in the database that belongs to the class, the use of **a** demands that one be created. After the proposition

WOMAN #1 is a woman

is asserted, the term **a woman** evaluates to **WOMAN #1**, which then appears as an argument of the proposition being asserted.

<18> Display the woman.
WOMAN #1

Another article, **the**, is similar to **a** in that it evaluates to the first element that can be generated as the instance of a given class, but **the** generates an error if no such element exists.

<19> Deny any man does like a woman.
<20> Assert any man does like any woman.
<21> Assert John does not like Sara.
<22> ?

[GLOBAL Database]
JOHN DOES NOT LIKE SARA.
ANY MAN DOES LIKE ANY WOMAN.
SARA IS A WOMAN.
MARY IS A WOMAN.
JOHN IS A MAN.

<23> Redo 7.
MARY

Now, returning to our original definition of **does like**, lines <20> and <21> show how we can establish a default relation and an exceptional relation, respectively. In line <23>, we use the **redo** procedure to re-execute line <7>, we see that only **Mary** can be generated as a member of the given class. When attempting to prove '**John does like Sara**' ROSIE hits the proposition asserted in line <21> before the one from line <20>, thus disproving rather than confirming the target proposition and filtering **Sara** out of the class **woman that John does like**.

<24> Activate beliefs.
<25> Assert John does like Sara.
<26> ?
[BELIEFS Database]
JOHN DOES LIKE SARA.

ROSIE allows users to create alternate databases and bring them into context with the **activate** procedure seen in line <24>. Here we create and activate the database **beliefs**. Unless otherwise directed, assertions go into the active database, and so the assertion in line <25> went into **beliefs**.

<27> Global?

```
[ GLOBAL Database ]
  JOHN DOES NOT LIKE SARA.
  ANY MAN DOES LIKE ANY WOMAN.
  SARA IS A WOMAN.
  MARY IS A WOMAN.
  JOHN IS A MAN.
```

<28> Display every woman.

SARA

MARY

<29> Redo 7.

SARA

MARY

Line <27> demonstrates another monitor command for examining the contents of a particular database; note that the information in the global database was not lost even though this database is no longer active. Line <28> demonstrates yet another attribute of ROSIE's database structure, namely, that information stored in the global database, even when not active, is still accessible.⁴ Also notice that when we re-execute line <7>, we now get both **Sara** and **Mary** again. This is because ROSIE can prove '**John does like Sara**' in the active database before finding it could be disproved in the global database.

<30> Clear database.

<31> ?

```
[ BELIEFS Database ]
```

<32> Redo 7.

MARY

Another way to remove propositions from a database is with the **clear** procedure. Now when we re-execute line <7>, **Sara** is again filtered from the given class.

3.3 BUILDING ROSIE PROGRAMS

One builds a ROSIE program out of *rules*, *rulesets*, and *demons*. These constitute the principal programming structures in ROSIE (they are discussed in more detail in Chapter 4). Basically, a rule is an executable programming statement (a monitor rule is simply a rule issued at the top-level monitor), while a ruleset is a rule subroutine; demons are special types of rulesets. To start a program, one issues a rule to the top level that invokes a ruleset. The ruleset executes the rules in its body that will presumably make changes to the database, query the user for information, and/or invoke other rulesets, etc.

⁴This is only true of the global database.

Rulesets cannot be defined at the top level, but must be defined in *program files*. Program files can also contain rules *not* contained in a ruleset; these are called *file rules*. File rules are normally used to initialize the database. Although a single program file can contain any number of rulesets and file rules, it is a good practice to organize large applications into several files: one for the main body of code, one for utility rulesets, and one for file rules. Program files are created and further changed via operations of the file package (see Chapter 13).

Program files are stored on disk. To incorporate an existing file into ROSIE, use the **load** procedure, e.g.,

```
<2> Load "myprog".
```

This tells ROSIE to load the program file "**myprog**" from disk.⁵ Loading does two things: first, it *notices* the contents of the file; and second, it *enables* those contents. Noticing is the act of recording the file's contents, while enabling is the process of defining rulesets and executing file rules. A file can also be enabled without being noticed via the **sysload** procedure. Program files that you wish to edit during a ROSIE session should be loaded--you can only edit noticed files--while other program files should be sysloaded to make efficient use of space.

Program files are created with the **build** procedure, e.g.,

```
<3> Build "players".
```

Build creates a noticed (albeit empty) program file of the given name. To add rulesets and file rules, use the **edit** procedure, e.g.,

```
<4> Edit "players".
```

This calls up the user's preferred text editor⁶ on the ROSIE edit file (i.e., a buffer file, which, in UNIX based ROSIE, is called **.rosie-ed**). When the user exits the editor, ROSIE parses and loads the contents of the edit file, retaining it in place of the original code.

To see how this works, assume we called the editor as in line <4>, added some code and exited the editor, e.g.,

⁵Actually, ROSIE loads the file **myprog.map** or, if compiled, **myprog.cmp**.

⁶ROSIE determines the user's preferred editor from the LISP variable **\$ROSIEEDITOR**. In UNIX based ROSIE, this variable is initially set to the environment variable **EDITOR**, or just "edit" if this variable is not set.

Scanning...

Done.

Parsing...

TO FIND BASKETBALL PLAYERS
TO DECIDE IF A PERSON IS TALL
Done.

Loading...

TO FIND BASKETBALL PLAYERS
TO DECIDE IF A PERSON IS TALL
Done.

The **list** procedure allows us to examine the contents of the file, e.g.,

<5> List "players".

To find basketball players:

[1] Send "{Every man who is tall} is a basketball player.{cr}".
End.

To decide if a person is tall:

[1] If the person's height is greater than 6.7 feet,
conclude true, otherwise conclude false.
End.

[rule 1] Assert each of Jim, Jack, John, and Joe is a man.

[rule 2] Let Jim's height be 6.4 feet and
Jack's height be 6.8 feet and
John's height be 5.7 feet and
Joe's height be 7.1 feet.

As you can see this is a very simple program. The two file rules initialize the database to contain several men of varying heights, e.g.,

<6> ?

[GLOBAL Database]
7.1 FEET IS A HEIGHT OF JOE.
5.7 FEET IS A HEIGHT OF JOHN.
6.8 FEET IS A HEIGHT OF JACK.
6.4 FEET IS A HEIGHT OF JIM.
JOE IS A MAN.
JOHN IS A MAN.
JACK IS A MAN.
JIM IS A MAN.

The procedural ruleset '**to find basketball players**' defines the procedure that starts the program running, and the predicate ruleset '**to decide if a person is tall**' is used to decide when a particular candidate meets the requirement for being a basketball player. We would start this program by calling the **find basketball players** procedure, e.g.,

```
<7> Find basketball players.
JOE is a basketball player.
JACK is a basketball player.
```

which prints a message for each suitable candidate it finds.

To update the program, we simply call the editor on the program file and make the desired changes. While this is good for a small file, it seems an awkward approach for making a small change to a large file containing possibly several dozen rulesets. To combat such situations, ROSIE provides a special-purpose data primitive called a *filesegment*.

A filesegment allows the user to specify contiguous portions of a program file, such as a sequence of file rules or ruleset rules as well as an entire ruleset or program file. In addition, certain types of filesegments can be specified using a convenient shorthand notation, e.g., the filesegment

```
'file: "players"'
```

can be specified as simply "players" and

```
'file: "players", to decide if a person is tall'
```

as tall. The file package operations and break package operations take filesegments as arguments. Thus, the file package and break package operations can be applied to entire files or portions thereof, e.g.,

```
<8> List 'file: "players"'.

```

```
To find basketball players:
```

```
[1] Send "{Every man who is tall} is a basketball player.{cr}".
End.
```

```
To decide if a person is tall:
```

```
[1] If the person's height is greater than 6.7 feet,
      conclude true, otherwise conclude false.
End.
```

```
[rule 1] Assert each of Jim, Jack, John, and Joe is a man.
```

```
[rule 2] Let Jim's height be 6.4 feet and
          Jack's height be 6.8 feet and
          John's height be 5.7 feet and
          Joe's height be 7.1 feet.
```

```
<9> List 'file: "players", to decide if a person is tall'.
```

```
To decide if a person is tall:
```

```
[1] If the person's height is greater than 6.7 feet,
      conclude true, otherwise conclude false.
End.
```

<10> List tall.

To decide if a person is tall:

[1] If the person's height is greater than 6.7 feet,
conclude true, otherwise conclude false.

End.

<11> List 'file: "players", to decide if a person is tall, 1'.

[1] If the person's height is greater than 6.7 feet,
conclude true, otherwise conclude false.

If we wish to change the criterion for deciding tallness in the demo program, we could simply edit the ruleset defining this property, e.g.,

<12> Edit tall.

In fact, we could just edit the offending rule of that ruleset, e.g.,

<12> Edit 'to decide if a person is tall, 1'.

Scanning...

Done scanning.

Parsing...

Done parsing.

Loading...

TO DECIDE IF A PERSON IS TALL -- redefined.

Done loading.

<13> List tall.

To decide if a person is tall:

[1] If the person's height is greater than 6.9 feet,
conclude true, otherwise conclude false.

End.

After editing, the ruleset will be redefined and exhibit the desired new behavior, e.g.,

<14> Find basketball players.

JOE is a basketball player.

So far, everything we have done to "**players**" has been kept in core, i.e., nothing has been safely written to disk. Edits can be written to disk with the **save** procedure. Given a program file, **save** updates the program's **.txt** and **.map** files to reflect changes made during a ROSIE session. If, as in the case of "**players**", the files **players.txt** and **players.map** do not yet exist, they are created. The **.txt** file is a copy of the program source code, and the **.map** file is its **HILEV** representation as well as a mapping between source and **HILEV**.

Once satisfied with a program's behavior, the user can improve performance of the program file via the **compile** procedure. Compiling a program file creates a **.cmp** file containing the binary machine code representation of the **HILEV** in the **.map** file. Rulesets run on the order of three to five times faster when compiled.

3.4 DEBUGGING FACILITIES

If not satisfied with a program's behavior, ROSIE provides several facilities for monitoring various aspects of its behavior. These facilities can be used alternately for finding bugs in the code or improving performance.

The primary source of debugging aids come from the *break package*. The *break package* (discussed in Chapter 14) allows the user to temporarily redefine rulesets and demons in order to monitor control flow and/or interrupt execution at key points. There are three basic facilities in the *break package*: a *trace facility*, *break facility*, and *profile facility*. The *trace facility* redefines rulesets such that a message is printed before and after invocation. The *break facility* temporarily interrupts execution, throwing control into an interactive *break loop* from which system state can be examined and the computation aborted or resumed. The *profiler* redefines rulesets to collect performance information on each invocation. When, via the *break package*, an errant ruleset has been found, it is then a simple matter to edit and fix the bug.

Of course, if the bug was introduced by a misinterpretation of ROSIE's syntax, it may be very hard to find. That is, the errant code may look correct, but its syntactic interpretation is not what one might think. This type of problem occurs most frequently with rules of associativity, e.g., is

record the name of the ship

really

record (the name of the ship)

which is the intuitive interpretation, or

record (the name) of the ship

which is ROSIE's interpretation. Such problems can be detected with the **deparse** procedure.

The deparser (part of the file package) is a ROSIE source code generator. Given the HILEV representation of a filesegment, the deparser generates its source code equivalent. While this generated code lacks certain stylistic eloquence--the deparser is not a very good pretty printer--it does highlight ROSIE's interpretation of code by delimiting possibly ambiguous code segments with parentheses and illustrating associations of embedded code blocks with indentation.

For instance, consider the procedural ruleset

To report a finding:

[1] Send "{the finding}{cr}".

[2] Add the proposition from "{the finding}" to findings.

End.

adapted from the SPILL demo program. In this ruleset, the finding is a string such as

"spill is detected at WOC-6"

which reports some observation about a chemical spill as a ROSIE proposition. Rule [2] of the ruleset is supposed to turn the finding into an *intentional proposition* and assert it into the findings database. When we call this ruleset, e.g.,

```
<16> Report "spill is detected at WOC-6".
      spill is detected at WOC-6
```

```
In 'TO REPORT A FINDING, AT 2'
No such element exists:
THE PROPOSITION
```

```
Broken at:
      'TO REPORT A FINDING, [rule] 2'.
```

```
[1] quit.
```

```
In 'TO REPORT A FINDING, AT 2'
```

```
<17>
```

an error occurs, throwing control into a break loop. Examining this ruleset with the deparser, e.g.,

<17> Deparse report.

TO REPORT A FINDING:

[1] SEND {THE FINDING, CR}.

[2] ADD (THE PROPOSITION) FROM {"", THE FINDING, ""} TO FINDINGS.
END.

we see that from "{the finding}"⁷ actually modifies **add** rather than **the proposition**. We can fix this problem by delimiting the correct interpretation with parentheses, e.g.,

<18> Edit report.

Scanning...

Done scanning.

Parsing...

TO REPORT A FINDING

Done parsing.

Loading...

TO REPORT A FINDING -- redefined.

Done loading.

<19> List report.

To report a finding:

[1] Send "{the finding}{cr}".

[2] Add (the proposition from "{the finding}") to findings.

End.

<20> Deparse report.

TO REPORT A FINDING:

[1] SEND {THE FINDING, CR}.

[2] ADD (THE PROPOSITION FROM {"", THE FINDING, ""}) TO FINDINGS.
END.

<21> Report "spill is detected at WOC-6".

spill is detected at WOC-6

<22> Findings?

[FINDINGS Database]

SPILL IS DETECTED AT WOC-6.

and the ruleset now runs to completion.

⁷Lexical analysis transforms this into the expression seen in the example.

3.5 ERRORS, INTERRUPTS, AND BREAK LOOPS

ROSIE is very tolerant of errors and other interrupts. Most runtime errors are recoverable, allowing computations to resume from the point of the error. When a recoverable error occurs, control is thrown into an interactive monitor, called a *break loop*, from which the interrupted computations can be resumed. User interrupts, signaled by hitting <ctrl>C, are also treated as recoverable errors.

A break loop is an extension of the top-level monitor. The extensions primarily reside in a set of break commands that are accessible only within a break loop. Details about all the break commands can be found in Chapter 14. Briefly, the break commands allow the user to examine various aspects of system state at the time of the break and then resume computations from that point.

For instance, consider our previous example using the ruleset

To report a finding:

[1] Send "{the finding}{cr}".

[2] Add the proposition from "{the finding}" to findings.

End.

When we ran this the first time, e.g.,

```
<16> Report "spill is detected at WOC-6".
spill is detected at WOC-6
```

```
In 'TO REPORT A FINDING, AT 2'
```

```
No such element exists:
```

```
THE PROPOSITION
```

```
Broken at:
```

```
'TO REPORT A FINDING, [rule] 2'.
```

```
[1]
```

we encountered a runtime error due to the fact that **the proposition** was being evaluated instead of **the proposition from "{the finding}"**.

Since this is a recoverable error, control is thrown into a break loop.⁸ If you do not wish to deal with the error, you can return to the top level via the **quit** procedure as we did before. This aborts the computation. Otherwise, you can try to fix the error and resume computations from the break point.

⁸If this error had not been recoverable, such as a *stack overflow error*, control would have returned to the top-level monitor.

The prompt in a break loop is the line number surrounded by square brackets ([]). Square brackets are used to remind you that the current environment resides *inside* the invocation of a ruleset. Note that each break loop has its own command history and, thus, each break loop starts at line [1]. Also note that you can have several layers of break loops, e.g.,

```
[1] Report "spill is not detected at WOC-6".
spill is not detected at WOC-6
```

```
In 'TO REPORT A FINDING, AT 2'
No such element exists:
THE PROPOSITION
```

```
Broken at:
'TO REPORT A FINDING, [rule] 2'.
```

You can return to earlier breaks with the **quit!** break command, e.g.,

```
[1] Quit!
```

```
Broken at:
'TO REPORT A FINDING, [rule] 2'.
```

```
[2]
```

which is distinguished from the **quit** procedure by terminating with an exclamation point (!).

From the break loop, we can examine aspects of system state, such as the state of the invocation's private database, e.g.,

```
[2] Private?
[ PRIVATE Database ]
"spill is detected at WOC-6" IS A FINDING.
```

We can also execute anything we can execute from the top level, e.g.,

```
[3] Deparse report.
```

```
TO REPORT A FINDING:
[1] SEND {THE FINDING, CR}.
[2] ADD (THE PROPOSITION) FROM {"", THE FINDING, ""} TO FINDINGS.
END.
```

```
[4] Edit report.
```

allowing us to discover and repair the problem. Once fixed, we can resume computations from the broken rule, e.g.,

```
[5] Resume!
```

or from some other rule, e.g.,

```
[5] Resume 1!
```

or simply restart the ruleset invocation, e.g.,

```
[5] Resume ruleset!
```

There are also break commands such as **list!** and **edit!** for easily examining and editing the broken ruleset rule and commands such as **trace!** and **pop!** for examining and moving about the stack of ruleset invocations. For further details, see Chapter 14.

3.6 EXITING A ROSIE SESSION

Always terminate a ROSIE session with the **logout** procedure, e.g.,

```
<25> Logout.
```

```
Files edited but not saved:
```

```
'FILE: "players"'
```

```
Save 'FILE: "players"' (Y or N)? y
```

```
Saving 'FILE: "players"'...
```

```
Done saving.
```

As the example above demonstrates, **logout** performs various clean-up tasks, such as closing open files, ending a dribble session, or informing the user of edits not written to disk.

When running under UNIX, the user can temporarily suspend a ROSIE session by hitting <ctrl>Z; this returns the user to the operating system level. The session can be resumed in the same manner as any other suspended UNIX job (i.e., typically by typing **fg**).

3.7 SYSTEM SWITCHES

ROSIE supports a small number of *system switches* to control certain aspects of system behavior (these switches are implemented as LISP variables whose values are **T** when *on* and **NIL** when *off*). Some switches are supported to make ROSIE 3.0 act like earlier versions of ROSIE and others simply to suppress noncritical features that the user may not like.

The system switches supported by ROSIE 3.0 are further described in Appendix C.

3.8 TOP-LEVEL OPERATIONS

The following are operations found to be useful at the top level. The file package commands are not included here but can be found in Chapter 13.

?

Lists the contents of the active database; equivalent to executing the **show** procedure, e.g.,

```
<2> Assert each of Jim, Jack, and John is a man.
<3> ?
[ GLOBAL Database ]
  JOHN IS A MAN.
  JACK IS A MAN.
  JIM IS A MAN.
```

<name element>?

<name element> ::= [<atom>]* <atom>

Lists the contents of the database named <name element>. Equivalent to calling **show** <name element>, e.g.,

```
<4> Assert 'John does love Mary' in active beliefs.
<5> Active beliefs?
[ ACTIVE BELIEFS Database ]
  JOHN DOES LOVE MARY.
```

??

Lists up to the last 40 monitor rules issued by the user, e.g.,

```
<6> ??
<6> ??
<5> ACTIVE BELIEFS?
<4> ASSERT 'JOHN DOES LOVE MARY' IN ACTIVE BELIEFS.
<3> ?
<2> ASSERT EACH OF JIM, JACK AND JOHN IS A MAN.
<1> DRIBBLE TO "log".
```

<integer>?

Lists the rule designated by <integer> if it was one of the last 40 monitor rules issued by the user, e.g.,

<7> 2?

<2> ASSERT EACH OF JIM, JACK AND JOHN IS A MAN.

redo a line [thru a line] [for N times]

Re-executes the designated sequences of monitor rules, where each *line* must evaluate to a positive integer specifying one of the last 40 monitor rules, e.g.,

```
<8> Redo 3.
[ GLOBAL Database ]
  JOHN IS A MAN.
  JACK IS A MAN.
  JIM IS A MAN.
```

If the **for N times** option is given, re-executes the sequence *N* times, where *N* must be a positive integer.

redo

Re-executes the previous line.

fix [a line]

Allows users to edit and resubmit monitor rules. Calls the user's text editor on a dummy file containing the text of the designated rule, executing the modified rule when the user exits the edit session. The new rule is remembered instead of the call to **fix** that exhumed it, and thus, the new rule can be referred to later, e.g.,

```
<9> Fix 2.

<9> ASSERT EACH OF MARY AND SARA IS A WOMAN.
<10> ??

<10> ??
<9> ASSERT EACH OF MARY AND SARA IS A WOMAN.
<8> REDO 3.
<7> 2?
<6> ??
<5> ACTIVE BELIEFS?
<4> ASSERT 'JOHN DOES LOVE MARY' IN ACTIVE BELIEFS.
<3> ?
<2> ASSERT EACH OF JIM, JACK AND JOHN IS A MAN.
<1> DRIBBLE TO "log".
```

If no *line* is given, applies **fix** to the previous line.

describe an element [in a database]

Displays all propositions from *database* that use *element* as a top-level argument, e.g.,

<11> Describe John in active beliefs.

JOHN DOES LOVE MARY

forget about an element [in a database]

Removes all propositions from *database* that use *element* as a top-level argument, e.g.,

<12> Forget about John in active beliefs.

<13> Active beliefs?

[ACTIVE BELIEFS Database]

display an element

Prints *element* to standard output (by default, the user's terminal), e.g.,

<14> Display the woman.

SARA

dribble to a file

Opens a special output channel to *file*, making it the dribble file. After this, a copy of all terminal I/O will be sent to *file*. The dribble file may not be closed except with **stop dribbling**.

This is a convenient way to save a transcript of all or part of a ROSIE session for later viewing.

NOTE: You may edit while dribbling, but that part of the session will *not* be dribbled.

stop dribbling

Closes the dribble file and stops copying terminal I/O. If no dribble file is open, an error occurs.

switch on a switch**switch off a switch**

Respectively, enables or disables *switch*, which is one of ROSIE's

system switches defined in Appendix C, e.g.,

<15> Switch on \$MIXPRINTMODE.

<16> ?

[GLOBAL Database]

SARA is a woman.

MARY is a woman.

JOHN is a man.

JACK is a man.

JIM is a man.

NOTE: When the \$MIXPRINTMODE is on, ROSIE uses the older form of printing relations in which keywords are in lowercase and everything else (sometimes) is in uppercase.

toggle on *a switch*

toggle off *a switch*

Like **switch on/off** except that, if executed in a ruleset, *switch* is reset to its original value when the ruleset terminates.

toggle *a switch*

If *switch* is on, turns it off, otherwise turns it on.

a switch **is set**

Concludes true if *switch* is on, concludes false otherwise, e.g.,

<17> If \$MIXPRINTMODE is set, display yes.

YES

info switches

Lists the setting of all system switches, e.g.,

<18> Info switches.

\$AUTOQUERYFLG is off

\$COMPRULESETS is off

\$EXPDRULESETS is on

\$EXTENDSEARCH is on

\$MIXPRINTMODE is on

\$PRETTYFORMAT is off

\$PRINTMSGs is on

\$REMOVEDUPLS is off

info system

Lists the system name, version number, the version of LISP used to implement ROSIE, and the system creation date, e.g.,

<19> Info system.

```
(R)
[ ROSIE      Version 3.0  (PSL)  15-Apr-86 ]
```

info date

Prints the current date and system name in comment form, e.g.,

<20> Info date.

```
(R)
[ ROSIE      Version 3.0  (PSL)  30-May-86 ]
```

info loaded

Lists pertinent information about loaded and noticed program files, e.g.,

<21> Info loaded.

Files currently loaded and noticed by filepkg:

```
'FILE: "report"'
  Last changed: Not written to disk
  Not compiled
```

```
  Contains 1 ruleset, 0 filerules
  All rulesets enabled
```

```
  Edited but not saved
```

```
'FILE: "players"'
  Last changed: Fri May 30 14:11:31 1986
  Not compiled
```

```
  Contains 2 rulesets, 2 filerules
  All rulesets enabled
```

type *a file*

List the contents of *file* on the user's terminal.

delete a file

Deletes *file* from the user's directory. Does not ask for confirmation. No error occurs if the file does not actually exist.

dskin a file

Loads *file* using the implementation LISP's **load** function (or its equivalent). Provided for loading LISP files into the system.

NOTE: For further information on the entire set of operations for file input and output, see Chapter 11.

lisp

Throws control into a LISP break loop. Exiting the break loop (in PSL this is done by hitting <ctrl>D or **q**) returns control to the top-level monitor.

paraphrase

Throws control into a special monitor for verifying the interpretation of ROSIE code.

The parsemode monitor reads and parses ROSIE rules and other file items, and then displays its interpretation of the code using parentheses and indentation to highlight the boundaries of terms, clauses, and phrases, e.g.,

```
<22> paraphrase.  
> Move the ship from the port.  
MOVE (THE SHIP) (FROM THE PORT).  
  
> Move (the ship from the port).  
MOVE (THE SHIP (FROM THE PORT)).  
  
>Quit.  
<23>
```

Exiting this monitor by hitting a <ctrl>C or by entering **quit**, throws control back to the top-level monitor.

parsemode

Like **paraphrase**, this operation throws control into a special monitor for examining the interpretation of ROSIE code. However, the

parsemode monitor displays the HILEV interpretation of the code, e.g.,

```
<23> parsemode.
> Move the ship from the port.
(<RULE>
  (<DO>
    (<GO> (<IDENT> MOVE
      (*OBJECT* (<THE> (<DESC> <D/56236/G0383>
        (<IDENT> SHIP) NIL NIL)))
      (FROM (<THE> (<DESC> <D/56253/G0384>
        (<IDENT> PORT) NIL NIL)))))))))

> Move (the ship from the port).
(<RULE>
  (<DO>
    (<GO> (<IDENT> MOVE
      (*OBJECT* (<THE>
        (<DESC> <D/58293/G0386>
          (<IDENT> SHIP
            (FROM
              (<THE>
                (<DESC> <D/58276/G0385>
                  (<IDENT> PORT) NIL NIL))))
          NIL NIL)))))))))

> Quit.
<24>
```

HILEV is the machine executable representation of a parsed file item.

reclaim

Forces a LISP garbage collect (i.e., reclaims previously used dynamic storage that is no longer needed so that it may be allocated for another purpose later on).

save as a file

Writes the current ROSIE session core image as an executable file to *file*. This essentially "freezes" the state of the program that calls it. The user can run the executable file and resume the ROSIE session following the point at which **save** was called.

NOTE: **Save as** closes all open channels (including the channel to the dribble file) before creating the executable file.

logout

Kills the ROSIE session. THIS IS THE ONLY SAFE WAY TO EXIT ROSIE.

If edited program files have not been written to disk, ROSIE will issue a warning and allow the user to save his edits using the file package operation **save**.

IV. PROGRAMMING STRUCTURES

ROSIE's principal programming structures are *rules*, *rulesets*, and *demons*. Rules correspond to executable programming statements, while rulesets equate to rule subroutines; demons are a specialized form of ruleset. ROSIE programs are defined as collections of interacting rulesets and demons. To run a program, one issues a rule to ROSIE's top-level monitor, which immediately executes the rule. The rule will invoke a ruleset, which executes the rules in its body, invoking other rulesets, and so on.

In this chapter, we provide a definition of rules and rulesets, primarily focusing on how one defines and invokes rulesets.

4.1 RULES

`<rule> ::= <action block> .`

`<action block> ::= <action>
 ::= <action> AND <action block>`

Rules consist of a sequence of one or more *actions*,¹ separated by the conjunctive **and** and terminated by a period (**.**), e.g.,²

**Assert the report was received at the current time and
relay that report to every module.**

**If any red battalion does advance toward any strategic
objective and that objective is undefended,
move some blue battalion to that objective and
report 'that battalion was directed to that objective'.**

**For each blue battalion (BBTL) in sector #15,
advise BBTL to 'move to Red River Crossing' and
assert BBTL was given a new directive.**

**While any strategic objective is not defended,
keep some blue battalion on alert.**

¹The concept of "one or more action" appears frequently and is referred to as an *action block*.

²The first example rule contains two actions, an *assert action* and a *procedure*; the second, a *conditional action*; the last two example rules illustrate two different types of *iterative actions*. Note that the conditional and iterative actions take nested action blocks as arguments.

A rule executes each of its component actions in turn. Execution halts after the last action. Execution can be terminated earlier by executing a terminating procedure (i.e., **return**, **produce**, **conclude**, or **continue**), or by aborting computations with the **quit** procedure.³

4.2 RULESETS

```

<ruleset> ::= <header> :
            [<private decl>]
            [<monitor decl>]
            [<rule>]*
            <end stmt>

            ::= SYSTEM RULESET <header> :
            <system body>
            <end stmt>

<header> ::= TO GENERATE <genr form>
           ::= TO DECIDE [IF] <pred form>
           ::= TO <proc form>

<genr form> ::= [<spec>] <root name> [<private pps>]

<proc form> ::= <atom> [<formal>] [<private pps>]

<pred form> ::= <formal> <be aux> <a/an> <root name> [<private pps>]
              ::= <formal> <be aux> <atom> [<formal>] [<private pps>]
              ::= <formal> <be aux> <prep> [<formal>] [<private pps>]
              ::= <formal> <do aux> <atom> [<formal>] [<private pps>]

<spec> ::= ( | THE | A | AN | )

<root name> ::= <atom> [<atom>]*

<private pps> ::= <prep> <formal> [<private pps>]

<formal> ::= [<a/an>] <root name>

<private decl> ::= PRIVATE [:] <class list> .

<class list> ::= <formal> [( [INITIALLY] <term> )] [, <class list>]

<monitor decl> ::= EXECUTE SEQUENTIALLY .
                 ::= EXECUTE CYCLICALLY .

```

³From the programmer's standpoint, rules are not all that interesting. They are essentially a linguistic convenience used to string together groups of actions. Actions are what actually control system behavior and are discussed further in Chapter 5.

```

::= EXECUTE RANDOMLY .

<system body> ::= (LAMBDA (-args-) . -body-)

<end stmt> ::= END .

```

The applicability and context in which rules are executed can be controlled by grouping rules into *rulesets*. Like subroutines in more conventional programming languages, rulesets provide a convenient way to modularize rules into coherent procedural units. One of ROSIE's strengths is that these modules can be invoked in a natural and transparent way using generalized English-like linguistic structures.

Users can define three different types of rulesets: *procedural*, *predicate*, and *generator* rulesets. Each type varies along the lines of how it is invoked and what values (if any) it can return. There also exists a hybrid class of rulesets called *demons*. Demons fire upon the occurrence of some specified event and have the capability of deciding whether to let the interrupted event proceed. Rulesets and demons can optionally be designated as *system rulesets*. The body of a system ruleset is defined using a LISP *lambda form*, which provides a convenient mechanism to interface with software packages developed in or accessible from LISP.

4.2.1 Defining Rulesets

Rulesets (and demons) are defined in *program files*; ROSIE does not allow rulesets to be defined from the top-level monitor.⁴ A ruleset definition begins with a *header* statement, which must be terminated by a colon (:), and ends with an *end* statement. After the header, the user may optionally specify a declaration of *private classes* and an *execution monitor*. The body of a ruleset is defined as an ordered sequence of zero or more rules.⁵

As an example, consider the ruleset,

⁴For more information on building and loading program files, see Chapter 13.

⁵A ruleset consisting of zero rules is a null-op.

```

To generate a response to a query:
Private: a reply.
Execute cyclically.
[1] Send "{cr}{the query} ".
[2] Read "{anything (bind the reply)}.{cr}".
[3] If the lowercase of the reply = either "yes" or "no",
    produce that reply,
    otherwise send "Please enter YES or NO.{cr}".
End.

```

This is a generator ruleset, so designated by the fact that its header starts with the words

To generate . . .

This ruleset produces instances of the class **response to query**, where *query* could be anything (but will probably be a *string element* posing a yes/no question). It uses the private class **reply** and a cyclic execution monitor. Its body consists of three rules, numbered [1] through [3]. The behavior of this ruleset is to prompt the user repeatedly with the value of **the query** until the user inputs **yes** or **no**.

4.2.1.1 Header Statements

A ruleset's header designates its type, names its formal parameters and identifies its calling form. There are three basic forms a header can have. These correspond to the three types of rulesets as well as ROSIE's three principal linguistic structures. A precise description of these three forms will be given in Section 4.2.2. In this section, we will examine the basics.

A ruleset's type is implicit in the syntax of its header. The header of a generator ruleset always begins with the keywords

To generate . . .

while a predicate ruleset header always begins with

To decide . . .

and a procedural ruleset header simply begins with

To . . .

While the exact syntactic structure of headers varies between ruleset types, they all share a few fundamental aspects.

The header always specifies the name of the ruleset. This will be one (or possibly more) token(s)⁶ corresponding to the root concept being defined. For instance, the name of the procedural ruleset

⁶Only generator rulesets and predicate rulesets that test class relations (i.e., relations using the *is-a* copula) can have multitoken names.

To move *a ship* from *a source* to *a destination*:

is **move**, the name of the predicate ruleset

To decide if *a ship* was anchored in *a port*:

is **anchored**, and the name of a generator ruleset

To generate the absolute value of *a number*:

is **absolute value**.

Another common feature of headers is the specification of formal parameters. These are called *private classes* because they are specified as indefinite references to a class and are bound and accessed accordingly (in the above examples the private classes are italicized). Private classes fill distinct linguistic roles. For instance, both procedural and predicate rulesets can optionally take a parameter that acts as the object of their main verb, and a predicate ruleset always has a parameter that plays the role of its subject. Additional private class parameters can be specified in a chain of prepositional phrases⁷ appearing at the end of the header. Such a chain can contain any number of preposition/private class pairs as long as the same preposition is not used twice.

The calling form of a ruleset is derived from its type, name, associated prepositions, and additional type-specific features. Note that the private class parameters of a header have no bearing on the unique identification of a ruleset. This means that you may not have two rulesets defined simultaneously if they differ only in their parameter names. Note also that, internally, prepositions are alphabetized. This means that the order of prepositional phrases in the calling form does not have to correspond to the order found in the header.

Essentially, a ruleset's header identifies the fixed and variable components of syntax of the ruleset calling form. The fixed parts include the ruleset's name and all prepositions. The variable parts consist of the private classes. All fixed parts *must* appear verbatim in the calling form. The user passes arguments by replacing the variable parts with specific arguments.

⁷Since private classes may not be modified by a prepositional phrase, there is no prepositional attachment ambiguity in ruleset headers.

4.2.1.2 Private Class Declarations

The user may optionally specify additional private classes with a private class declaration. Private classes are normally used like local variables in other programming languages and are discussed further in Section 4.2.4.

The syntax of a private class declaration is defined as

```
<private decl> ::= PRIVATE [:] <class list> .  
<class list> ::= <formal> [( [INITIALLY] <term> )] [, <class list>]
```

Example--

Private: a counter (initially 0), a reply.

If given, this declaration must appear immediately after the ruleset header.

Note that it is possible to specify a initial binding for a private class. When the ruleset is invoked, the initial values are evaluated and bound sequentially, and later bindings *may* reference earlier.

4.2.1.3 Execution Monitors

The user may also optionally designate the manner in which rules are to be executed with an execution monitor declaration, the syntax of which is defined as

```
<monitor decl> ::= EXECUTE SEQUENTIALLY .  
                  ::= EXECUTE CYCLICALLY .  
                  ::= EXECUTE RANDOMLY .
```

This declaration tells ROSIE how to execute the rules in the ruleset's body, i.e., with either a *sequential*, *cyclic*, or *random* monitor.

The sequential monitor executes rules one at a time, terminating the ruleset invocation after the last. The cyclic monitor executes sequentially, but starts over again after executing the last rule. The random monitor repeatedly executes a rule selected by a pseudo-random number generator.

A monitor declaration must appear after the private class declaration (if given) and before the first rule of the ruleset, and there can be only one monitor in a ruleset. The monitor defaults to sequential.

4.2.1.4 The Ruleset Body

The body of the ruleset can be any sequence of rules. The file package automatically inserts line numbers in the form of comments (e.g., [1]) before the first line of every rule. The file package maintains these numbers and updates them whenever the ruleset is edited or otherwise modified.

4.2.1.5 End Statements

The end statement is simply the keyword **end** followed by a period (.). All rulesets must be terminated by an end statement. If the file package encounters a ruleset that is not terminated by an end statement, it will warn the user and automatically insert the missing end.

4.2.2 Ruleset Types

There are three types of rulesets: *procedural*, *predicate*, and *generator* rulesets. Each ruleset type serves a conceptually different purpose, each gets invoked in a different way, and each returns a different form of value.

4.2.2.1 Procedural Rulesets

```
<header> ::= TO <proc form>
```

```
<proc form> ::= <atom> [<formal>] [<private pps>]
```

A procedural ruleset enacts a *procedure* (a type of action) and does not return a value to the calling form. As an example, consider the procedural ruleset,

```
To move a vessel from a source to a destination:  
[1] Deny the vessel is docked at the source.  
[2] Assert the vessel is docked at the destination.  
End.
```

which updates the database when invoked by a procedure such as in

```
Move USS Nimitz from Le Havre to Auckland.
```

Procedural rulesets allow users to define conceptually modular tasks that can be parameterized conveniently.

The syntax of the header for procedural rulesets is shown above. In this syntax, <atom> plays the role of an imperative verb and names the ruleset (e.g., **move** in the example). A procedural ruleset can take an optional direct object that can be followed by any number of

preposition/private class pairs. The calling form for a procedural ruleset is identified by its name, whether it takes a direct object and its associated prepositions.

When a procedural ruleset invocation terminates, it returns control to the calling form. The invocation of a procedural ruleset can be terminated in one of two ways. If it is executing its rules under a sequential monitor, then it terminates after the last rule. It can also be terminated by executing the **return** procedure.

4.2.2.2 Predicate Rulesets

```

<header> ::= TO DECIDE [IF] <pred form>

<pred form> ::= <formal> <be aux> <a/an> <root name> [<private pps>]
               ::= <formal> <be aux> <atom> [<formal>] [<private pps>]
               ::= <formal> <be aux> <prep> [<formal>] [<private pps>]
               ::= <formal> <do aux> <atom> [<formal>] [<private pps>]

<root name> ::= <atom> [<atom>]*

<be aux> ::= WAS [NOT]
           ::= WERE [NOT]
           ::= AM [NOT]
           ::= ARE [NOT]
           ::= IS [NOT]
           ::= WILL [NOT] BE

<do aux> ::= DID [NOT]
           ::= DO [NOT]
           ::= DOES [NOT]
           ::= WILL [NOT]

```

A predicate ruleset provides a means of computing the truth or falsity of a *proposition* (a declarative n -ary relation). When ROSIE cannot otherwise decide a proposition's *truth value* from relations in its database, it automatically invokes the corresponding predicate ruleset if one exists. For instance, the predicate ruleset

```

To decide if a vessel is seaworthy:
[1] If the vessel does float, conclude true,
    otherwise, if the vessel does leak,
    conclude false.
End.

```

will be invoked by

```

If USS Nimitz is seaworthy,
    move USS Nimitz from Le Havre to Auckland.

```

if the proposition '**USS Nimitz is seaworthy**' cannot otherwise be proved or disproved from assertions in the database. A predicate ruleset can conclude true or false, returning a boolean value to the calling form, or it can simply terminate, returning nothing and implying an indeterminate truth value.

There is some degree of variation in the syntax of predicate ruleset headers; this corresponds to the five syntactic forms of propositions (see Section 6.2.1). Predicate rulesets that test a class relation can have a multitoken name (i.e., <root name>) that represents the base class for which the test holds, e.g., the name of

To decide if a target is a strategic target in a zone:

is **strategic target**. The other predicate types are named by a single token representing the main verb of the clause; this can either be an <atom>--e.g., the name of

To decide if a target is located in a zone:

is **located**--or a <prep>--e.g., the name of

To decide if a target is on a hill:

is **on**. The calling form of a predicate ruleset is identified by its name and associated prepositions, its tense and auxiliary verb form, and whether it takes a direct object.

Note that while one can define a predicate ruleset (with the **not** option) that tests the negation of propositions, the presence or absence of this syntax is not used in calling or storing predicate rulesets. Thus, a predicate and its negation may not be defined simultaneously, and whichever is defined would decide the truth or falsity of a proposition or its negation.

ROSIE will attempt to invoke a predicate ruleset whenever it cannot decide the truth or falsity of a proposition by a search of the *physical database* (see Chapter 10). If a predicate ruleset is defined that matches the proposition (i.e., contains the same main verb, prepositions, auxiliary form, and tense) then it is invoked, and its conclusion determines the result of the test. If the predicate or the proposition is negated, then the inverse of the conclusion decides the result.

The conclusion of a predicate ruleset can be one of true, false, or indeterminate. If the invocation is terminated by executing the **conclude** procedure, its conclusion will be either true or false. If terminated by executing **return** or by running out of rules to execute, it concludes indeterminate and the truth or falsity of the proposition goes undecided.

4.2.2.3 Generator Rulesets

<header> ::= TO GENERATE <genr form>

<genr form> ::= [<spec>] <root name> [<private pps>]

<spec> ::= (| THE | A | AN |)

A generator ruleset procedurally defines a *class* of elements, producing instances of the class one-by-one on demand. Like predicate rulesets, they are invoked indirectly through interactions with the database.

A generator ruleset may be invoked whenever a request is made for instances of a class. As described in Chapter 7, this happens when a description is used as a generator. When generating from a class (say, the class of *ship*), ROSIE first produces all elements from assertions in the database that satisfy the proposition '*element is a class*', e.g., assuming the database contains

USS Nimitz is a ship
 USS Coral Sea is a ship
 USS Enterprise is a ship

then generating every ship successively produces USS Nimitz, USS Coral Sea, and USS Enterprise. Once such elements have been exhausted, ROSIE can invoke a generator ruleset for computing additional members of the class. For instance, the generator ruleset

To generate a vessel at a port:
 [1] Produce every boat which is docked at the port.
 [2] Produce every ship which is docked at the port.
 End.

would produce a continuous stream of elements when invoked by

While some vessel at Auckland is not seaworthy,
 repair that vessel.

until all elements produced satisfied the '*element is seaworthy*' predicate. Note that an invocation of a generator ruleset does not necessarily produce a single instance of a class; it can produce zero or more instances.

Instances are produced from a generator ruleset using the **produce** procedure. This procedure takes a single argument that it *produces* as an instance of the generator's class. If the element produced does not satisfy the restrictions posted by the calling form, or if more than one element is requested, then control is passed back to the ruleset. Execution continues from the point where the element was produced. A single invocation of a generator ruleset can execute the **produce** procedure many times, generating a stream of elements.

An invocation of a generator ruleset is terminated when an element produced satisfies halting conditions of the calling form, or the **return** procedure is executed, or (under a sequential monitor) the last rule is executed.

A generator ruleset is named by the root name of its class; this can be one or more tokens in length. The calling form for a generator is its name and associated prepositions. Unlike procedural and predicate rulesets, generator rulesets may not take an optional direct object argument.

4.2.3 Invoking Rulesets

As illustrated in the last section, different types of rulesets are invoked in different ways. Procedural rulesets are invoked by executing a procedure, predicate rulesets are invoked when testing a proposition, and generator rulesets are invoked when generating instances of a class. Demons, which are discussed in the next section, are invoked just prior to the occurrence of an event. Regardless of these differences after being called, the invocation of any type of ruleset always proceeds in the same fashion.

First, the invocation is given a *private database*. The ruleset's formal parameters (i.e., the private classes taken from its header) are bound in the private database to the values of the actual parameters of the calling form. Then the rules in the ruleset's body are executed according to the ruleset's *execution monitor*. A ruleset invocation can be terminated in one of three ways. If the execution monitor is sequential, then the invocation can terminate upon executing the last rule. Executing a *terminating procedure* is another, somewhat more standard way to exit a ruleset, and executing the **quit** procedure, which aborts computation and throws control back to the top-level monitor, is a third method.

4.2.3.1 Calling Forms

A calling form identifies the target ruleset by its name and associated prepositions, e.g., the procedure

Move USS Nimitz from La Havre to Auckland.

is a calling form for the procedural ruleset

To move a ship from a source to a destination:

Note that ROSIE does not use the formal parameters of the ruleset header to decide which ruleset to invoke, only the ruleset's type, keyword(s) and associated prepositions. Additionally, the order of prepositional phrases does not matter. Thus, for instance, the procedure

Move UCS Nimitz to Auckland from La Havre.

will behave identically to the one seen above.

4.2.3.2 Argument Passing

Arguments are passed to a ruleset invocation via the private classes in the ruleset's header. Before the call, the actual parameters of the calling form are evaluated. For each value, a proposition of the form

actual is a formal

is asserted into the private database, where *actual* is the value of the actual parameter and *formal* is the corresponding private class. For instance, in our previous example, i.e.,

Move USS Nimitz from La Havre to Auckland.

the propositions,

**USS Nimitz is a ship
La Havre is a source
Auckland is a destination**

would be asserted into the private database of the ruleset invocation.

4.2.3.3 The Private Database

Each ruleset invocation is provided with a private database for storing the intermediate results of computation. The contents of the private database can be accessed only from within that invocation (i.e., a private database can be thought of as lexically scoped). Once the invocation terminates, the private database and its contents are discarded.

As compared to the global database and alternate databases (see Chapter 10), a private database is a considerably restricted form of storage. These restrictions include the following:

- Only class membership (i.e., *is-a*) relations may be stored in a private database;
- There may be at most one instance of a private class at any given time, e.g., if **counter** is a private class, then

Assert 1 is a counter and 2 is a counter.

will replace 1 with 2 as the single instance of **the counter**;

- If not appearing among the formal parameters of the header, private classes can be declared only in a private class declaration, e.g.,

Private: a counter (initially 0), a result.

- Private classes may not be modified by a prepositional phrase or relative clause, e.g., the term

the counter which is greater than 10

will not be treated as reference to a private class even if **counter** is a private class.

- An instance of a private class may be generated using only the function word **the**, e.g., **a counter** and **every counter** will ignore the fact that **counter** is a private class;
- Manipulations of the private database will *not* invoke demons (i.e., demons only monitor manipulations of the global and active database).

The instance of a private class (i.e., its value) can be accessed with the function word **the** and changed with the database actions **assert**, **deny**, and **let**. While this discussion may make private classes and the private database appear rather complex, you'll actually find their use to be quite intuitive and the restrictions negligible.*

4.2.3.4 Execution Monitors

The order of execution of rules in the ruleset's body is determined by the ruleset's execution monitor. This can be one of *sequential*, *cyclic*, or *random*. The sequential monitor executes rules one at a time from first to last, terminating the invocation after the last rule. The cyclic monitor repeatedly executes the rules from first to last. The random monitor repeatedly executes rules drawn at random.

4.2.3.5 Terminating Procedures

If a ruleset's execution monitor is sequential, then the invocation terminates after the last rule. If it is not, then the invocation must be explicitly terminated by executing one of the terminating procedures.* The following operations can be used to terminate a ruleset invocation:

*Readers already familiar with ROSIE should note that the restrictions on private classes are new; existing code may require editing to reflect these changes.

*Termination can also be achieved by executing the **quit** procedure, which aborts computation and throws control back to the top level.

return

Terminates the invocation of a ruleset or demon.

In procedural rulesets, simply ends call. In generator rulesets, causes the ruleset to stop producing elements. In predicate rulesets, concludes an indeterminate truth value. In demons, causes interrupted event to be discarded.

produce an element

This procedure may be executed only within a generator ruleset or a produce demon.

In a generator ruleset, produces *element* as an instance of the class being generated. Returns control to the invocation if *element* does not satisfy a halting condition established by the calling form. Otherwise, terminates the invocation.

In a produce demon, continues the produce event, substituting *element* for the element being produced.

NOTE: Due to frequency in use, the **produce** procedure has a special syntax. Normally, a procedure will take all possible prepositional attachments, e.g., the call

Move the ship from the port.

is interpreted as

Move (the ship) from the port.

where **move** takes the longest chain of prepositional phrases possible. To avoid the over use of parentheses to delimit its single argument, **produce** will try to take *no* prepositional attachment. Thus, the call

Produce the ship from the port.

is interpreted as

Produce (the ship from the port).

rather than

Produce (the ship) from the port.

This syntax was provided as a convenience and can be overridden by explicitly delimiting arguments with parentheses.

conclude true
conclude false

Terminates the invocation of a predicate ruleset, concluding either true or false for the truth value of the proposition being tested.

continue

Terminates a demon invocation, informing ROSIE to resume computations of the interrupted event (i.e., the event that triggered the demon).

quit [*because a string*]

Throws control to the top-level monitor. If the **because** option is given, *string* is printed to the standard output channel (normally the user's terminal) before aborting.

4.3 DEMONS

<header> ::= BEFORE EXECUTING <proc form>

::= BEFORE TESTING [IF] <pred form>

::= BEFORE GENERATING <genr form>

::= BEFORE PRODUCING <genr form>

::= BEFORE ASSERTING <pred form>

::= BEFORE DENYING <pred form>

<genr form> ::= [<determiner>] <root name> [<private pps>]

<proc form> ::= <atom> [<formal>] [<private pps>]

<pred form> ::= <formal> <be aux> <a/an> <root name> [<private pps>]

::= <formal> <be aux> <atom> [<formal>] [<private pps>]

::= <formal> <be aux> <prep> [<formal>] [<private pps>]

::= <formal> <do aux> <atom> [<formal>] [<private pps>]

Demons are a hybrid class of rulesets that allows users to selectively capture control of execution just prior to the occurrence of an event. As such, demons provide a mechanism for event-driven program control. They can be used for tracing and debugging during the program development. They can monitor changes to the database and check for consistency as the database undergoes change. Once invoked, a demon can decide whether or not the operation it preempted should resume.

For example, consider the demon

Before executing to move a ship from a source to a destination:

[1] Unless some vessel at the source is equal to the ship,
return, otherwise continue.

End.

which would be awakened by

Move USS Nimitz from Le Havre to Auckland.

Execution of the procedure would continue only if its arguments (i.e., USS Nimitz, Le Havre, and Auckland) satisfy the constraints set by the demon.

When a demon is defined, it establishes a process that monitors the initiation of an event. Only certain events can be monitored by demons; these include:

- 1) the execution of a procedure
- 2) the assertion, denial and testing of a proposition
- 3) a request to generate elements of a class
- 4) the production of a generated element.

Upon the initiation of its event, a demon is invoked. At this point, the demon can interrogate the system state and either allow the interrupted event to continue or release control without continuing the event.

4.3.1 Types of Demons

There are six types of demons, which may, respectively, monitor six types of events. These include:

- *procedural demons*, for monitoring the execution of procedures; e.g.,

Before invoking to move a ship from a source to a destination:

- *assert, deny, and test demons*, for monitoring assertions, denials, and tests of propositions;

Before asserting a ship is docked at a port:

Before denying a ship is docked at a port:

Before testing a ship is docked at a port:

- *generate demons*, for monitor requests to generate elements from a class;

Before generating a vessel at a port:

- *produce demons*, for monitoring when an element is actually produced as an instance of a class; e.g.,

Before producing a vessel at a port:

A demon's type and the exact situation it monitors is determined by its header. As one can observe, the header syntax for demons closely resembles that for procedural, predicate, and generator rulesets. Likewise, the naming conventions are also the same.

The database demons (i.e., assert, deny, and test demons) share a restriction not found in predicate rulesets. That is, the truth value of the target proposition *must* match the truth value indicated in a demon's header, otherwise the demon will not be invoked. Where a predicate ruleset may be invoked upon testing some relation or its negation, the database demons strictly monitor operations on the relation their header specifies. Thus, for example, to trap the assertion of a proposition and its negation, two assert demons must be defined.

4.3.2 Demon Invocation

A demon is invoked immediately before the event it monitors is about to occur. For instance, an assert demon will be invoked immediately before a proposition that it identifies is to be affirmed in the physical database.

Once invoked, the demon follows the standard invocation procedure of any ruleset. The arguments of the calling form are asserted into a private database, and the rules in the demon's body are executed under either a sequential, cyclic, or random monitor. When the invocation terminates, the interrupted event is either continued or discarded.

The only way to resume the interrupted event is to terminate the demon invocation with the **continue** procedure. Termination by any other means (with one exception that we will see in produce demons) causes the event to be discarded, returning control back to the rule that initiated the event.

4.3.3 The Generator Demons

The generator demons (i.e., generate and produce demons) may seem redundant at first. However, they demonstrate the distinct difference between requesting that elements be generated from a class and actually producing those elements. For example, the term **every man** issues a single request to *generate* the elements of the class **man**, which will one-by-one *produce* every instance of that class. A generate demon for the class **man** would be invoked once by **every man** (i.e., prior to issuing the generate request), while a produce demon for this class would be invoked zero or more times (i.e., once for every member of the class).

A generate demon is similar to the other types of demons. It is invoked immediately before the occurrence of a generate event and upon termination can either continue the event or not. If it continues the event, it has no effect on the elements produced.

A produce demon, however, can have great effect on the elements produced. When a produce demon is invoked, its private database will contain an *extra* relation not specified in its header or private class declaration. This extra relation will be of the form

element is a root name

where *element* is the element about to be produced and *root name* is the name of the demon; e.g., in the demon

Before producing a gauge reading:

this would be **gauge reading**. This allows the demon to consider the element when deciding whether to continue the produce event. If the invocation terminates with **continue**, then the given element is produced as an instance of the class. If terminated by **return**, then the element is not produced.¹⁰

Unlike other demons, the invocation of a produce demon can also be terminated with the **produce** procedure. When executed in a produce demon, **produce** continues the interrupted event, substituting its argument for the original element being produced.

As an example application of the produce demon, assume that we have a generator ruleset that produces a stream of real numbers in the range of -1.0 to 1.0 for the class **gauge reading**. Now assume that we want only the absolute value of these numbers. One way to effect this change is explicitly generate the absolute value of each **gauge reading**, e.g.,

Display the absolute value of every gauge reading.

but this method is somewhat awkward and verbose. Another way to effect the change is with a produce demon defined as

Before producing a gauge reading:

[1] Produce the absolute value of the gauge reading.
End.

which will continue every produce event with the absolute value of the private class **gauge reading**, the element originally being produced.

¹⁰This does not abort the generate event, it simply means that element will not be produced.

4.3.4 The Error Demon

Automatic error recovery can be controlled through the use of a special assert demon called *the error demon*. When processing a recoverable error (see Chapter 12 and Appendix B), ROSIE simulates an assertion of the proposition

<string, filesegment> is an error

where *<string, filesegment>* is a *tuple element* (see Section 9.4) containing *string*, which identifies the error message, and *filesegment*, which identifies the ruleset rule causing the error.

This proposition is not actually asserted into the database, but it will invoke an assert demon of the form

Before asserting a message is an error:

if such a demon exists. Further, if the error demon executes the **continue** procedure, then computation will be resumed automatically at the point of the error call.

4.4 SYSTEM RULESETS

```
<ruleset> ::= SYSTEM RULESET <header> :
              <system body>
              <end stmt>

<system body> ::= (LAMBDA (-args-) . -body-)

<end stmt> ::= END .
```

Rulesets and demons can optionally be designated as *system rulesets*. The body of a system ruleset is defined as a LISP *lambda form*.¹¹ System rulesets are designated by putting the words

System ruleset . . .

directly before the ruleset header, e.g.,

```
System ruleset to decide if an element is an integer:
(lambda (elt) (cond ((fixp elt) '<true>)
                    (t '<false>)))

End.
```

Note that the only thing that can appear between the header and the end statement is the lambda form (i.e., no comments, rules, or

¹¹A working knowledge of LISP is assumed on the part of the reader.

declarations). System rulesets provide a convenient way to interface with software packages developed in or accessible from LISP.

4.4.1 Defining System Rulesets

Any ruleset (including demons) can be designated as a system ruleset by preceding its header with the words **System ruleset**. The body of a system ruleset is described as a LISP *lambda form*. That is, an *s-expression* of the form

(lambda (*-args-*) *-body-*)

where *-args-* is a sequence of zero or more atoms that act as the formal parameters of the lambda form, and *-body-* is a sequence of s-expressions that define the actions of the lambda form.

As a special word of caution, be careful to ensure that each system ruleset header is free of syntax errors, and, likewise, that its body is free of parentheses errors. The tokenizer scans for the body of a system ruleset by invoking the LISP reader upon recognizing a system ruleset header. If the header is syntactically incorrect, then the tokenizer will not know to invoke a LISP reader, resulting in the body of the system ruleset being scanned as a normal file item. As well, if the body has unbalanced parentheses, the LISP reader may read too far or not far enough. Any of these situations can cause much confusion and grief.

To avoid these and other difficulties with system rulesets, we recommend that all system rulesets of any ROSIE program be kept simple and stored in a separate program file from the program's non-system rulesets.

4.4.2 Calling System Rulesets

System rulesets are called in the same way that non-system rulesets are called. System rulesets can enact procedures, make conclusions about the truth or falsity of propositions, produce elements of a class, and interrupt and continue events. System rulesets *cannot* invoke other rulesets, nor can they access the ROSIE database.

Unlike normal rulesets, a system ruleset invocation does not receive a private database. Arguments are passed via the formal parameters of the system ruleset's lambda form. When called, the lambda form associated with the system ruleset is applied (i.e., using LISP's **apply** function) to arguments of the calling form. The actual parameters are ordered by their linguistic role. The subject (if a predicate) comes first, then direct object (if any), and then the objects of prepositions. The objects of prepositions are ordered alphabetically by their preposition; they are *not* arranged in the order in which they appear in the ruleset header or the calling form.

4. Programming Structures

87

The following rules apply when trying to emulate standard terminating procedures from various types of system rulesets:

- To emulate the **return** procedure, simply return the value **NIL** from the lambda form.
- To emulate the **conclude** procedure, return one of **<true>** or **<false>** as the value of the invocation.
- To emulate the **produce** procedure, the ruleset must return either an atom, such as an *id*, *number*, or *string*, or a list. An atom is treated as a single element to be produced, while a list is treated as a sequence of such elements.

To produce a list of elements as a tuple element, ROSIE provides the function **list-to-tuple (lst)**, which coerces *lst*, a list of elements, into a tuple; embedded lists will also be coerced into tuples.

- To emulate the **continue** procedure, the demon should return the atom **<continue>**.

Additionally, if a produce demon system ruleset returns a non-NIL value, this value will be produced in lieu of the original value. Note also that a produce demon system ruleset receives an extra argument (i.e., the value to be produced) as the last parameter in its calling form. Thus, the formal parameters in its body should have one argument more than the private class specified in its header.¹²

¹²Numerous examples of system rulesets can be found in ROSIE's System Ruleset Library.

V. ACTIONS AND CONTROL FLOW

In this section, we describe an important aspect of any programming language, its control structures. In ROSIE, most control structures are implemented as *actions*, which constitute a principal syntactic and linguistic category. Actions can be used to invoke a procedure, to conditionally execute or iterate over a block of actions, or to iterate actions over the instances of a class. Actions are also used to modify the contents of the database.

5.1 ACTIONS AND ACTION BLOCKS

```

<action> ::= <procedure>
          ::= <data action>
          ::= <cond action>
          ::= <cond block>
          ::= <iter action>
          ::= ( <action block> )

<action block> ::= <action> [AND <action>]*

```

Actions equate to executable operations. They are stand-alone statements that need not be used as an argument to anything to be understood or to be executed. Actions are the things that start a ROSIE program in motion and keep it going.

ROSIE provides a variety of action types. Action syntax was designed to read like English and to indicate the type of operation an action performs. Procedures, which are user-defined actions, can be equally readable when their names and parameters are selected carefully.

A few action types take blocks of actions as parameters. For example, the *conditional action*, i.e.,

```
if <condition>, <action block>, otherwise <action block>
```

takes two *action blocks* (as well as a *condition*) as arguments. An action block is a sequence of one or more actions separated by the conjunctive **and**.

5.1.1 Types of Actions

There are five types of actions¹ in ROSIE. These five action types offer substantial power and flexibility for encoding aspects of control

¹Earlier releases supported an additional action type, *execute actions*, using **go** and **call**. Because the functionality of *execute actions* is subsumed by the *procedure* action type and the *intentional*

and considerable variation in the manner in which such code can be written. Briefly, these action types include:

- 1) *Procedures*, which are denoted by an imperative verb, optionally followed by a direct object and a chain of prepositional phrases, e.g.,

report the finding to the strategic command post

The behavior of a procedure is defined by a corresponding procedural ruleset; procedures are essentially "user-defined" actions.

- 2) *Database actions*, which enact changes to the database, e.g.,

assert the finding was reported at time 100
deny the finding is not substantiated
let the counter be the counter + 1
create an airfield

- 3) *Conditional actions*, which are *if-then-else* style actions, e.g.,

if the finding was reported and that finding was verified,
 deny the finding is unsubstantiated and
 act upon that finding,
otherwise verify the finding

unless the counter is greater than or equal to 0,
 produce the negation of the counter

- 4) *Conditional blocks*, which correspond to *case* statements in other programming languages, e.g.,

select the country:
 <Russia, Cuba>
 display bad guys;
 <USA, England>
 display good guys;
 <any third world nation>
 display cant tell;
default: display need more info

Conditional blocks consist of an ordered sequence of *key/action block* pairs. They execute the first action block whose associated key satisfies some selection criteria.

- 5) *Iterative actions*, which conditionally iterate over an action block, e.g.,

 while the counter is less than the upper bound,
 let the counter be the counter + 1 and produce the counter

or which iterate over the instances of a class, e.g.,

 for each target at the airfield,
 rate that target for each of capacity and vulnerability

and execute an action block on each iteration.

5.1.2 Associativity of Action Blocks

As demonstrated above, several action types take action blocks as arguments. Because of this, potential ambiguity arises in regard to nested action blocks. The rule defining the associativity of action blocks, as well as the precedence of constructs taking action blocks as arguments, is quite simple:

Actions associate with the most deeply nested action block not otherwise delimited (and, thus, closed to association).

In other words, action blocks are right associative, and constructs that take action blocks as arguments are of equal precedence.

Consider the following dummy rule

Execute action #1 and
execute action #2 and
if condition #1 is true,
 execute action #3 and
 execute action #4,
otherwise,
 execute action #5 and
 for each description #1,
 execute action #6 and
 execute action #7.

In this example, there are four action blocks. Indentation illustrates the association of otherwise ambiguously embedded blocks.

5.1.3 Comma Blocks and Parentheses

If the default associativity is undesirable, action blocks can be delimited using parentheses and commas. Action blocks delimited by commas are called *comma blocks*. The appeal of comma blocks over parenthesized action blocks is that of readability and naturalness; commas are a far more English-like delimiter. However, comma blocks can be used only in certain situations; parentheses are a more obvious and versatile delimiter.

Any action block can be delimited by surrounding it with a matching set of left and right parentheses. Since this form of delimiting is actually provided by the grammar rule

```
<action> ::= ( <action block> )
```

it might be more appropriate to say that an action can be an action block surrounded by parentheses. Were we to restate our previous example as

```
Execute action #1 and
execute action #2 and
(if condition #1 is true,
  execute action #3 and
  execute action #4,
otherwise,
  execute action #5) and
(for each description #1,
  execute action #6) and
execute action #7.
```

the new association would significantly change the interpretation of the rule. Unfortunately, parentheses are a somewhat awkward and un-English-like delimiter; their overuse can give otherwise readable ROSIE code the sometimes frightening appearance of LISP.

Comma blocks allow programs to avoid the overuse of parentheses. Essentially, a comma block is an action block introduced by a comma (,) and terminated by a comma, a period, a closing parenthesis, or (in some situations) a semicolon. For instance, the above example could be rewritten as

```
Execute action #1 and
execute action #2 and
if condition #1 is true,
  execute action #3 and
  execute action #4,
otherwise,
  execute action #5, and
for each description #1,
  execute action #6, and
execute action #7.
```

and it would have exactly the same interpretation.

Comma blocks can appear only in actions that take action blocks as arguments. This is to say, comma blocks can appear only within conditional actions, iterative actions, and conditional blocks;² refer to Sections 5.4, 5.5, and 5.6 for more details.

²These action forms practically required the use of comma blocks (or parentheses) to delimit their action block arguments. This has not

5.2 PROCEDURES

```

<procedure> ::= <atom> [<term>] [<pphrase>]
              ::= DO NOTHING

```

A procedure invokes a procedural ruleset identified by <atom>, which is treated as an imperative verb, the existence of a direct object, and the set of associated prepositions, e.g., the procedure

move USS Nimitz from Le Havre to Auckland

would invoke the procedural ruleset,

To move a vessel from a source to a destination:

if such were defined, passing the direct object (USS Nimitz) and objects of prepositions (Le Havre and Auckland) as arguments to the ruleset.

The **do nothing** procedure is defined specially and is intended to be used as a null-op, i.e., it does nothing. While this procedure has only limited utility in ROSIE 3.0, it is available in earlier releases of the language and is included in this release for compatibility.

5.3 DATABASE ACTIONS

```

<data action> ::= ASSERT <prop block>
               ::= DENY <prop block>
               ::= LET <let block>
               ::= CREATE <a/an> <description>

<prop block> ::= <proposition> [AND <proposition>]*

<let block> ::= <let form> [AND <let form>]*

<let form> ::= THE <description> BE <term>
             ::= <term> ' S <description> BE <term>
             ::= <term> BE THE <description>
             ::= <term> BE <term> ' S <description>

```

The database actions are used to add and remove relations from the database and are described in more detail in Chapter 10, which discusses ROSIE's database mechanism at length. Although they have the general appearance of procedures, their special argument types (i.e., *propositions* and *descriptions*, as opposed to terms) restrict them from being defined with procedural rulesets and necessitate that they be hardwired into the ROSIE grammar.

been the case in earlier versions of ROSIE and may be a possible source of syntactic incompatibility with existing code.

5.3.1 ASSERT... and DENY...

```

<data action> ::= ASSERT <prop block>
               ::= DENY <prop block>

```

```

<prop block> ::= <proposition> [AND <proposition>]*

```

Assert and **deny** respectively add and remove propositions from the database. They initiate an assert or deny event and can trigger an assert or deny demon.

Both **assert** and **deny** can take as arguments one or more propositions grouped over the conjunctive **and**, e.g.,

```

assert M6-1 is a child of WOC-6 and
      M6-2 is a child of M6-1 and
      M6-3 is a child of M6-1

```

They successively initiate an assert or deny event for each such proposition.

5.3.2 LET...

```

<data action> ::= LET <let block>

```

```

<let block> ::= <let form> [AND <let form>]*

```

```

<let form> ::= THE <description> BE <term>
             ::= <term> ' S <description> BE <term>
             ::= <term> BE THE <description>
             ::= <term> BE <term> ' S <description>

```

Conceptually, **let** is ROSIE's assignment operator. **Let** makes the value of <term> the singular instance of <description> in the database, e.g.,

```

let the counter be 1

```

is equivalent to executing

```

deny every counter is a counter and assert 1 is a counter

```

When **let** asserts and denies propositions, it is actually initiating assert and deny events that can trigger assert and deny demons. While these demons cannot discontinue a call to **let**, they can block any assert or deny event initiated by that call.

NOTE: <term> and <description> can be arranged in any order, e.g.,

```

let Gortz Airfield be the name of the strategic target
let the strategic target's name be Gortz Airfield

```


In the case of

let the strategic target be the enemy airfield

the value of the **enemy airfield** will become the singular instance of the **strategic target**, i.e., interpreted syntactically as

LET THE <description> BE <term>

5.3.3 CREATE...

<data action> ::= CREATE <a/an> <description>

Create creates a *name element* by appending #N to the *class noun* of <description>--N, a positive integer associated with the class noun, is incremented by one for each element so created. This element is then asserted as an instance of <description>, e.g.,

create a happy man

is equivalent to

assert man #1 is a happy man

This action is normally used to establish prototypical instance of a class.

5.4 CONDITIONAL ACTIONS

<cond action> ::= IF <condition> <then part> [<else part>]
 ::= UNLESS <condition> <then part> [<else part>]

<then part> ::= , [THEN] <action block> [,]
 ::= (<action block>)
 ::= THEN <action>

<else part> ::= (| OTHERWISE | ELSE |) , <action block> [,]
 ::= (| OTHERWISE | ELSE |) (<action block>)
 ::= (| OTHERWISE | ELSE |) <action>

The conditional actions correspond to the basic *if-then-else* type actions found in most high-level programming languages. ROSIE supports two forms of conditional actions, namely, the standard **if** form and its inverse **unless**.

5.4.1 IF... and UNLESS...

If and **unless** conditionally execute a block of actions. They both take as arguments a *condition* (see Chapter 6), which is a boolean combination of sentences, an action block, representing their *then-part*, and, optionally, another action block, representing their *else-part*.

If executes the actions in its *then-part* if its condition evaluates to true, otherwise it executes the actions in its *else-part*. Conversely, **unless** executes the actions in its *then-part* if its condition does not evaluate to true, and otherwise executes its *else-part*.

5.4.2 Associativity

When conditional actions are nested, a singleton *else-part* will always be associated with the most deeply embedded conditional action. For instance, consider

```

If condition #1 is true,
  if condition #2 is true,
    execute action #1,
  otherwise,
    execute action #2.

```

where indentation illustrates the attachment of the singleton *else-part*.

5.5 CONDITIONAL BLOCKS

```

<cond block> ::= <select block>
              ::= <choose block>
              ::= <match block>

```

Conditional blocks provide a specialized form of conditional control flow. They are roughly analogous to the *case* and *select* statements found in other programming languages. There are three types of conditional blocks, each handling a different situation.

A conditional block represents a decision table of the form

```

block type selector:
  [key action block;]*
  [default: action block]

```

whose entries are *key/action block* pairs and which permit the specification of a *selector* and, optionally, a *default action block*. A conditional block executes the first action block whose key satisfies the selector, executing the default block if no key is acceptable.

5.5.1 SELECT...

```

<select block> ::= SELECT <term> :
                    [<tuple element> <action body> [;]]*
                    [DEFAULT : <action block> [;]]

```

Select blocks most closely resemble the *case* statements available in other languages. The *selector* is a term that can evaluate to any arbitrary element, and each *key* is a tuple of elements.

When executed, **select** successively evaluates each key until it finds one containing an element that satisfies the test

element = selector

If such a key is found, its associated action block is executed, and the call to **select** is terminated. If no such key exists, then the default action block is executed.

As an illustration of this, consider the call

```

select the country:
  <Russia, Cuba>
    display bad guys;
  <USA, England>
    display good guys;
  <any third world nation>
    display cant tell;
  default: display need more info

```

which alternatively displays **bad guys**, **good guys**, etc., depending upon the value of **the country**.

5.5.2 CHOOSE...

```

<choose block> ::= CHOOSE SITUATION :
                    [IF <condition> <then part> [;]]*
                    [DEFAULT : <action block> [;]]

```

Choose blocks are similar to LISP's *cond* statement or *elseif*-type constructs in other languages. A choose block specifies a set of mutually exclusive conditions, and, once the appropriate condition is found, executes the actions associated with that condition. Thus, the *keys* of a choose block are conditions, and the *selector* is the state of the system.

When executed, **choose** successively evaluates each conditional key until it finds one that evaluates to true. If such a key is found, its associated action block is executed, and the call to **choose** is terminated. If no such key exists, then the default action block is executed.

As an illustration of this, consider the call

```
choose situation:
  if the rate is greater than 70,
    display good capacity;
  if the rate does range from 40 to 70,
    display fair capacity;
  default: display poor capacity
```

which performs a series of mutually exclusive tests and provides a default action if no test succeeds.

5.5.3 MATCH...

```
<match block> ::= MATCH <term> :
                  [<pattern element> <action block> [;]]*
                  [DEFAULT : <action block> [;]]
```

Match blocks resemble select blocks with a twist. The twist is that match blocks use ROSIE's string pattern matcher to decide which action block to execute. The *selector* of a match block should evaluate to a *string element* (see Section 9.5); if it does not, it will be coerced into a string. The *keys* of a match block are *pattern elements* (see Section 9.6).

When executed, **match** successively compares the selector to keys using the pattern matcher until it finds a successful match. If such a key is found, its associated action block is executed and the call to **match** is terminated. If no such key exists, then the default action block is executed.

As an example, consider

```
match the reply:
  {"Yes"|"yes"}
    conclude true;
  {"No"|"no"}
    conclude false;
  default: send "Please reply Yes or No!"
```

which might be used to decide if the user gave a yes/no answer to some query.

5.5.4 Associativity

Conditional blocks are right associative. In such cases where embedded condition blocks are not otherwise delimited, *key/action block* pairs and *default action blocks* will associate with the most deeply embedded conditional block. The extent of *key/action block* pairs and *default action blocks* can be delimited with semicolons (;). Like the comma, a semicolon is a more natural and English-like delimiter than parentheses.

As an example, consider

```
Execute action #1 and
select selector #1:
  <key #1>
    execute action #2 and
    execute action #3;
  <key #2>
    select selector #2:
      <key #3>
        execute action #4 and
        execute action #5;
      default: execute action #6 and
        execute action #7.
```

where indentation demonstrates the default association of embedded blocks. Terminating the default action block with a semicolon after action #6 causes the expression to be interpreted as

```
Execute action #1 and
select selector #1:
  <key #1>
    execute action #2 and
    execute action #3;
  <key #2>
    select selector #2:
      <key #3>
        execute action #4 and
        execute action #5;
      default: execute action #6; and
        execute action #7.
```

If we want action #7 to associate with the highest level action block, it is necessary to delimit the highest level select block with parentheses, as in

```
Execute action #1 and
(select selector #1:
  <key #1>
    execute action #2 and
    execute action #3;
  <key #2>
    select selector #2:
      <key #3>
        execute action #4 and
        execute action #5;
      default: execute action #6) and
execute action #7.
```

Going back to our original example, to get the default action block to associate with the higher-level select block, it is necessary to delimit the nested select block with parentheses, as in,

```

Execute action #1 and
select selector #1:
  <key #1>
    execute action #2 and
    execute action #3;
  <key #2>
    (select selector #2:
      <key #3>
        execute action #4 and
        execute action #5);
  default: execute action #6 and
           execute action #7.

```

As a final example, by terminating the default action block of this form with a semicolon, as in,

```

Execute action #1 and
select selector #1:
  <key #1>
    execute action #2 and
    execute action #3;
  <key #2>
    (select selector #2:
      <key #3>
        execute action #4 and
        execute action #5);
  default: execute action #6; and
           execute action #7.

```

we again get action #7 to associate with the highest level action block.

5.6 ITERATIVE ACTIONS

```

<iter action> ::= FOR EACH <description>
                [WHILE <condition>]
                [UNTIL <condition>] ,
                <action block> [,]

                ::= FOR EACH <description>
                [WHILE <condition>]
                [UNTIL <condition>]
                ( <action block> )

                ::= WHILE <condition>
                [UNTIL <condition>] ,
                <action block> [,]

```

```

::= WHILE <condition>
      [UNTIL <condition>]
      ( <action block> )

::= UNTIL <condition> ,
      <action block> [,]

::= UNTIL <condition>
      ( <action block> )

```

Iterative actions are used to conditionally loop over an action block. If the **for each** option is specified, the action block is executed for each instance of a description. If the **while** option is given, its associated condition is tested before each iteration; iteration continues until this condition evaluates to false. Conversely, if the **until** options is specified, its associated condition is tested *after* each iteration, and iteration is terminated when this condition tests true.

For each, **while**, and **until** can be used in unison. In such cases, the first time any halting condition is realized, the action terminates. For instance, the action,

```

for each enemy warship
  while there is an unassigned aircraft,
  deploy that aircraft to that warship

```

will repeatedly execute the **deploy** procedure until either every **enemy warship** has been produced or there are no more **unassigned aircraft**.

5.6.1 FOR EACH...

For each iterates over all elements that can be generated from a given *description* (see Chapter 7). On each iteration, the generated element is bound to the *description variable* of that description and can be referenced *anaphorically* (see Section 8.5) within the embedded action block. For instance, the action,

```

for each enemy warship, attack that warship

```

iterates over all instances of the class **enemy warship**. That **warship** is an anaphoric reference to each instance produced.

5.6.2 WHILE... and UNTIL...

The **while** and **until** options control iteration over an explicitly stated condition. The **while** form tests its associated condition *before* each iteration and terminates execution if its condition evaluates to true. On the other hand, the **until** form tests its condition *after* each iteration and terminates execution when its condition evaluates to true.

The semantics of the **until** form are somewhat problematic because, syntactically, it must appear *before* the loop's action block, rather than after, as its behavior would imply. Even experienced users often forget that its condition is not tested until after the action block has been executed.

5.6.3 Associativity

The only question of associativity with embedded iterative actions arises from the use of action blocks (actually, comma blocks). Note that this problem will also occur with conditional actions nested in iterative actions, iterative actions nested within conditional actions, and conditional actions nested within other conditionals. Although the associativity of action blocks was discussed earlier, it will be reviewed here for completeness.

Action blocks are right associative. Otherwise ambiguous actions within embedded action blocks will always be associated with the most deeply nested action block.

For example, the expression

```
For each description #1,  
  execute action #1 and  
  while condition #1 is true,  
    execute action #2 and  
    until condition #2 is true,  
      execute action #3 and  
      execute action #4.
```

would be interpreted as indicated by the given indentation. The action blocks of the conditional and iterative actions can be delimited by commas. Thus, rewriting the expression as

```
For each description #1,  
  execute action #1 and  
  while condition #1 is true,  
    execute action #2 and  
    until condition #2 is true,  
      execute action #3, and  
      execute action #4.
```

would associate **action #4** with the next highest action block. However, for **action #4** to be associated with the top-most action block, the embedded actions must be delimited by parentheses. Either of the expressions,

For each description #1,
execute action #1 and
 (while condition #1 is true,
 execute action #2 and
 until condition #2 is true,
 execute action #3) and
execute action #4.

or

For each description #1,
execute action #1 and
while condition #1 is true,
 execute action #2 and
 (until condition #2 is true,
 execute action #3), and
execute action #4.

will suffice to achieve the desired interpretation.

VI. CONDITIONS, SENTENCES, AND PROPOSITIONS

After actions, the next largest linguistic units are *conditions*, *sentences*, and *propositions*. Unlike action and action blocks, which can stand alone as programming statements, conditions, sentences, and propositions may only appear as arguments of higher-level expressions, such as conditional actions or database actions. A condition is a boolean combination of sentences that occurs within the context of a test, such as in **if**, **while**, or **until** actions. A sentence describes a declarative relation whose truth can be decided from the database. A proposition is a generalized type of sentence describing an n-ary relation that can be added to or removed from the database and for which truth or falsity can be computed by a predicate ruleset.

6.1 CONDITIONS AND BOOLEAN CONNECTORS

```

<condition> ::= <disjunct> [, OR <disjunct>]*
              ::= <disjunct> [, AND <disjunct>]*

<disjunct>  ::= <conjunct> OR <disjunct>
              ::= <conjunct>

<conjunct>  ::= <primary> AND <conjunct>
              ::= <primary>

<primary>   ::= ( <condition> )
              ::= <sentence>

<sentence>  ::= <proposition>
              ::= <special form>

```

A condition consists of a boolean combination of sentences, each of which describes a relation on one or more data objects. Sentences are combined using the boolean connectors **and** and **or** to create composite logical predicates. Component logical groupings of sentences can be indicated with commas and parentheses.

6.1.1 Boolean Connectors

There are only two boolean connectors, the conjunctive **and** and the disjunctive **or**, but there are two forms that these connectors can take. In one form, the sentence preceding the connector is terminated with a comma (,), and in the other, it is not. We shall refer to the connectors in the first form as *comma-and* and *comma-or* and the second as simple **and** and **or**. There is a subtle but distinct difference in the rules of associativity and precedence concerning these two forms.

As an example, consider the action,

If condition #1 is true or
condition #2 is true and
condition #3 is true,
execute action #1.

which is interpreted as

If condition #1 is true or
(condition #2 is true and condition #3 is true),
execute action #1.

where parentheses indicate the logical grouping of sentences. Compare this to a similar-looking action,

If condition #1 is true or
condition #2 is true, and
condition #3 is true,
execute action #1.

in which commas are used; this would be interpreted as

If (condition #1 is true or condition #2 is true) and
condition #3 is true,
execute action #1.

These opposing forms were developed to offer a more natural and English-like alternative (as opposed to parentheses) to delimiting logical groups of sentences. Notice, however, that the grammar does not allow an arbitrary depth of nesting without the aid of parentheses. Chains of *comma-and*s and *comma-or*s cannot appear in the same condition unless such chains are delimited by parentheses. This is an artificial constraint introduced in ROSIE 3.0 in order to avoid conditions that might become overly complex or otherwise confusing to the human reader.

6.1.2 Associativity and Precedence

All boolean connectors are right associative. Of the four forms of connectors, **and** has higher precedence and binds sentences more tightly than **or**, which has higher precedence than both *comma-and* and *comma-or*. Since *comma-and* and *comma-or* cannot be mixed, their relative precedence is unimportant.

6.2 SENTENCES

<sentence> ::= <proposition>
 ::= <special form>

A sentence describes a declarative relation between data objects, the truth or falsity of which can be tested. There are two types of sentences: *propositions* (called *primitive sentences* in earlier ROSIE documents); and everything else, where the "everything else" category subsumes several special-case sentence forms that are hardwired into the definition of the language.

6.2.1 Propositions

```

<proposition> ::= <term> <verb phrase>

<verb phrase> ::= <be aux> <a/an> <description>
                ::= <be aux> <atom> [<term>] [<pphrase>]
                ::= <be aux> <prep> [<term>] [<pphrase>]
                ::= <do aux> <atom> [<term>] [<pphrase>]

<be aux> ::= WAS [NOT]
          ::= WERE [NOT]
          ::= AM [NOT]
          ::= ARE [NOT]
          ::= IS [NOT]
          ::= WILL [NOT] BE

<do aux> ::= DID [NOT]
          ::= DO [NOT]
          ::= DOES [NOT]
          ::= WILL [NOT]

```

A proposition describes a general n-ary relation whose truth or falsity can be asserted into or denied from the database as well as tested against the database. In addition, the truth or falsity of a proposition can be computed by a predicate ruleset. Propositions can specify a relation between objects, e.g.,

John does sell shoes in Baltimore

an attribute of an object, e.g.,

John is happy

and the class of an object, e.g.,

John is a salesman

Propositions can also be manipulated as data elements called *intentional propositions* (see Section 9.10). The rules for asserting, denying, and testing propositions are described further in Chapter 10 with a discussion of ROSIE's database mechanism.

6.2.1.1 Verb Phrases

Propositions basically consist of two components: (1) a term, which plays the role of subject; and (2) a verb phrase, which categorizes the proposition and identifies its arguments. ROSIE permits five basic categories of verb phrases. Each category captures (roughly) a specific class of English usage, including:

Class Membership¹

(|WAS|IS|WILL|) [NOT] [BE] <a/an> <description>

. . . **is a doctor who does make house calls**
 . . . **will not be a witness**
 . . . **was an individual with glasses**

Predication

(|IS|WAS|WILL|) [NOT] [BE] (|<atom>|<prep>|) [<pphrase>]

. . . **is outside for the evening**
 . . . **was not alone**
 . . . **will be late for dinner**

Complementation

(|IS|WAS|WILL|) [NOT] [BE] (|<atom>|<prep>|) <term> [<pphrase>]

. . . **is nuclear powered**
 . . . **was not really exciting to Mary**
 . . . **will be running rapidly toward the goal**

Intransitivity

(|DID|DOES|WILL|) [NOT] (|<atom>|<prep>|) [<pphrase>]

. . . **did not divide by 2**
 . . . **does eat with a fork**
 . . . **will not fuss**

Transitivity

(|DID|DOES|WILL|) [NOT] (|<atom>|<prep>|) <term> [<pphrase>]

. . . **does study computer science**
 . . . **will cook a steak for dinner**
 . . . **did not water the plants**

¹Although we categorize **was a** and **will be a** under class membership, only relations using **is a** are examined to determine class membership. Asserting **Mary will be a woman** will never result in **Mary** being generated as an instance of the class **woman**.

Verb phrases are always introduced by the *is-a* copula or by an auxiliary form of *be* or *do*. The negation of a verb phrase can be derived by inserting the reserved word **not** immediately after the introductory form of *be* or *do*, and all verb phrases can be expressed in either past, present, or future tense.²

6.2.1.2 Properties of Class Relations

Propositions that describe class relations (i.e., using the class membership verb phrase) inherit special properties because they specify the relationship of an object to a *description* (see Chapter 7). When the description is modified by a *relative clause* (see Section 7.2), such propositions actually represent a set of relations.

Basically, descriptions name a subclass of elements. A description is dominated by a reference to a *class* (see Section 7.1). The description can optionally be modified by a relative clause that posts restrictions on the elements of the class. Thus, the description forms a subclass of those elements.

When a description is modified by a relative clause, the relations of the relative clause act as additional sentences to assert, deny, and test. For instance, the proposition,

**John is a salesman who does sell shoes in
Baltimore and who is not happy**

actually specifies the three relations,

John is a salesman
John does sell shoes in Baltimore
John is not happy

If the above proposition were asserted, then all three relations would appear in the database; if it were denied, than all three forms would be removed from the database; and if it were tested, than all three would have to be true for the test to succeed.

Because the relative clause of a description can be any arbitrary condition, sometimes a class relation cannot be asserted or denied. To assert or deny a class relation, its associated relative clause may only contain propositions combined under conjunction. See Chapter 7 for further details.

²Tense is provided only to enhance expressibility. While ROSIE can distinguish between propositions that differ in tense, it does not provide any temporal semantics for, say, deciding that a proposition in past tense is true because the same proposition in present tense was true previously, etc.

6.2.1.3 Negation: NOT...

As indicated earlier, the *complement* of a proposition can be obtained by inserting the reserved word **not** immediately after the auxiliary in its verb phrase. (This is also called *negating* the proposition.) A negated proposition can be asserted, denied or tested, acting in all but a single case as one would intuitively expect.

The one exceptional situation occurs with respect to any class relation whose description is modified by a relative clause. In this case, the negation remains on the base *is-a* relation and does *not* carry over into the relations specified in the relative clause. For example, the proposition,

John is not a salesman who does sell shoes in Baltimore

essentially specifies the relations,

John is not a salesman
John does sell shoes in Baltimore

rather than

John is a salesman
John does not sell shoes in Baltimore

or

John is not a salesman
John does not sell shoes in Baltimore

either of which seem somewhat more consistent. There is no good argument for why this definition has been adopted other than that the latter two definitions, if actually intended, can still be achieved by making precise statements, i.e.,

John is a salesman who does not sell shoes in Baltimore

and

John is not a salesman who does not sell shoes in Baltimore

6.2.2 Special Sentence Forms

<special form> ::= THERE IS <how many> <description>
 ::= <term> <special vp>

<how many> ::= NO
 ::= <a/an>

```

      ::= JUST ONE
      ::= MORE THAN ONE

<special vp> ::= HAS <how many> <description>
              ::= <rel op> <term>

<rel op> ::= IS [NOT] EQUAL TO
           ::= IS [NOT] GREATER THAN [OR EQUAL TO]
           ::= IS [NOT] LESS THAN [OR EQUAL TO]

           ::= =
           ::= ~=
           ::= >
           ::= ~>
           ::= >=
           ::= ~>=
           ::= <
           ::= ~<
           ::= <=
           ::= ~<=

```

Besides propositions, ROSIE allows several special sentence forms whose truth or falsity may only be tested. The definition of these forms is hardwired into the language and cannot be changed by the user.

NOTE: Many special verb forms appearing in earlier releases of ROSIE have been reimplemented as predicate rulesets and added to the system ruleset library. These predicates are described where pertinent in the sections on string elements, pattern elements, intentional propositions, and file I/O.

6.2.2.1 EQUAL TO..., LESS THAN..., and GREATER THAN...

Element equivalence can be tested using the sentence forms,

```

<term> is [not] equal to <term>
<term> [~]= <term>

```

which can be written in a terse or expanded notation. Similarly, the numeric comparison forms,

```

<term> is [not] greater than [or equal to] <term>
<term> [~]>[=] <term>
<term> is [not] less than [or equal to] <term>
<term> [~]<[=] <term>

```

can also be specified in a terse or expanded style. Each of these sentence forms takes two terms as arguments and applies the indicated comparison operator to the values of these terms. The comparison operators are described in more detail in Sections 9.1 and 9.3, respectively.

6.2.2.2 THERE IS...

The remaining special sentence forms are provided as a convenience for testing the cardinality of a subclass of elements specified by a description. The forms

there is no <description>
there is <a/an> <description>
there is just one <description>
there is more than one <description>

examine the number of elements that can be generated from a description: The first form, e.g.,

**If there is no objection to the bill,
ratify that bill.**

evaluates to true if *no* elements can be generated; the second form, e.g.,

**While there is an unassigned aircraft,
dispatch that aircraft to the strike force.**

evaluates to true if at least one element can be generated; the third form, e.g.,

**If there is just one runway at some airfield,
strike that runway at that airfield.**

if only one element can be generated; and the fourth form, e.g.,

**Until there is more than one task force at the river,
stay on alert.**

evaluates to true if at least two elements can be generated. Note that these forms all initiate a generate event for the description (see Section 7.6.2) and, therefore, can be interrupted by a generator demon.

6.2.2.3 HAS...

There also exists a hybrid form of the above, i.e.,

<term> **has no** <description>
<term> **has** <a/an> <description>
<term> **has just one** <description>
<term> **has more than one** <description>

which is semantically equivalent to

there is no <description> of <term>
there is <a/an> <description> of <term>

there is just one <description> of <term>
there is more than one <description> of <term>

respectively, where the prepositional phrase **of** <term> would act as a modifier on the description's class, e.g.,

John has a girlfriend who is tall

would actually be treated as the sentence

there is a girlfriend of John who is tall

and

Gortz Airfield has more than one runway in use

as the sentence

there is more than one runway of Gortz Airfield in use

Therefore, **has** simply acts as a restricted, yet at times more readable, form of **there is**.

VII. DESCRIPTIONS AND CLASSES

```

<description> ::= <class> [<rel clause>]
                ::= such <atom> [<desc var>]

<class> ::= <root name> [( <desc var> )] [<pphrase>]

<root name> ::= [<atom>]* <class noun>

<class noun> ::= <atom>

<desc var> ::= <atom>

```

The concept of a *description* underlies much of ROSIE's data manipulation capabilities. A description is composed of a *class reference*, an optional *relative clause*, and a *description variable*. A description's class names some set of elements satisfying an *is-a* relation, while its relative clause acts as a filter on those elements--thus, a description represents some subset of the elements named by its class. The description variable provides a unit of temporary storage for caching instances of a description.

Descriptions are used by ROSIE in three ways:

- 1) testing an element for membership in the set being described
- 2) generating one, some, or all members of that set
- 3) adding or removing members from that set.

For instances, given the example description,

vessel at Le Havre which is seaworthy

we can write three rules that illustrate the uses of a description:

*If USS Nimitz is a vessel at Le Havre which is seaworthy,
then move USS Nimitz from Le Havre to Auckland.*

Display every vessel at Le Havre which is seaworthy.

Let USS Nimitz be the vessel at Le Havre which is seaworthy.

The elements named by a description are retrieved from the database on demand. Because of this, their representation is implicit in the description. Thus, the set of elements described can change as the database changes.

7.1 CLASSES

`<class> ::= <root name> [(<desc var>)] [<pphrase>]`

`<root name> ::= [<atom>]* <class noun>`

`<class noun> ::= <atom>`

The major component of a description is its *class*. A *class* names a set of elements that satisfies an affirmed class relation or that can be generated as instances of the class.¹

Syntactically, a class reference is specified as a sequence of one or more tokens, called its *root name*, optionally modified by a chain of prepositional phrases,² e.g.,

any target at the airfield
every strategic command center for the red army
the member of <A, B, C> at 1

where the objects of prepositions act as arguments to the class. The last token of a root name is called its *class noun* and is used in anaphoric references to a description, e.g.,³

that target
some such center
that member

A class names a set of elements against which another element can be tested for membership or from which individual elements can be generated. Testing and generation may possibly invoke predicate rulesets or generator rulesets, respectively.

7.1.1 Testing for Membership

To test whether an element belongs to a class (e.g., to test if **John** belongs to the class **man**), ROSIE tests if a proposition of the form

¹Not all elements that satisfy a class relation can be generated as an instance of the class and visa versa.

²For the time being, ignore the optional <desc var> component, as it is a part of the surrounding description and not a feature of a class.

³The first and last are examples of *anaphoric terms* (see Section 8.5), and the middle is an example of a *quantified descriptive term* (see Section 8.4) using an *anaphoric description* (see Section 7.4).

element is a class

as in

John is a man

is provably true. This test conducts a search of the *physical* and the *virtual database* (see Chapter 10) as follows:

- 1) All affirmed propositions of the form

elt is a class

and

elt is not a class

are examined in order of recency (i.e., the most recently asserted is examined first);

- 2a) If the proposition being examined is of the form

elt is a class

and *element = elt* is true, where = tests equivalence as described in Section 9.1.3, then the membership test succeeds.

- 2b) If the proposition is of the form

elt is not a class

and *element = elt*, then the test fails.

- 3) If no affirmed proposition can prove or disprove membership, then ROSIE tests the proposition using a predicate ruleset. If a predicate ruleset of the form

To decide if *elt* is a class:

as in

To decide if an element is a man:

or its complement

To decide if *elt* is not a class:

is defined, then this ruleset is invoked, and its conclusion alternately confirms or disproves the membership test.⁴

⁴If the predicate is defined and does not conclude true or false, then the test fails. If instead its complement is defined and returns an indeterminate value, the test succeeds. Both *cannot* be defined simultaneously.

This is the general method applied to testing all propositions and is further described in Chapter 10.

7.1.2 Generating from a Class

To generate an instance of a class (e.g., an instance of **man**), ROSIE also examines the contents of the physical and virtual database. Elements are produced in succession until one of these elements satisfies a halting condition.⁵ Generation of instances of a class behaves as follows:

- 1) As with testing for membership, all affirmed propositions of the form

elt is a class

and

elt is not a class

are examined in order of recency;

- 2) When a proposition is of the form

elt is not a class

as in

Mary is not a man

or

any woman is not a man

then *elt* is added to a list of things known *not* to be in the class, and control is returned to Step 1. Otherwise, when a proposition is of the form

elt is a class

as in

John is a man

then we proceed to Step 3;

- 3a) If *elt* is not a *class element*⁶ and if *elt* is not equivalent to any member of the list of things known not to be in the class,

⁵The halting condition is established by the construct that initiates the generate event and is described further in Section 7.6.2.

⁶A *class element* is characterized as a description introduced by the function word **any** (see Section 9.8).

element is produced, terminating the generation process if it satisfies the halting condition;

- 3b) If *elt* is a class element, e.g.,

any mortal is a man

then Step 3a is applied to each successive element generated from its associated description, i.e., in this case, instances of the class **mortal**;

- 4) If, after exhausting all such affirmed propositions, no element produced satisfies the given halting condition, ROSIE looks up a generator ruleset of the form

To generate a class:

as in

To generate a man:

If such a ruleset exists, it is invoked, and each element generated via the **produce** procedure is applied to Step 3a.

NOTE: When examining all affirmed class relations in Step 1 above, ROSIE is working with a *partially closed* database. This means that ROSIE will examine only those class relations that were affirmed at the beginning of the generate event. Thus, the action,

For each man, create a man.

will cause ROSIE to create a new **man** for each instance already in the database; it will not cause ROSIE to create instances of **man** indefinitely.

7.1.3 Potential Pitfall to Class Membership

One undesirable side effect to the definitions for membership and generation is a potentially contradictory enumeration of class members. Namely, the elements represented by a class may be determined by one of two methods: 1) finding all elements that satisfy a class membership test; or 2) finding all elements that can be generated as an instance of the class. Although these statements intuitively sound alike, they actually can describe two distinct sets of elements.

Proving that an element satisfies a class relation may invoke a predicate ruleset, while enumerating the elements of a class may invoke a generator ruleset. Since there is no practical method to ensure consistency in the definitions of corresponding rulesets, the two methods are not necessarily equivalent.

As an example, one may define the predicate

```
To decide if a person is a man:
[1] If the person is equal to either Jim, Jack or John,
    conclude true,
    otherwise, conclude false.
End.
```

which implicitly defines one set of elements for the class **man**, and then a generator ruleset

```
To generate a man:
[1] Produce each of Bill, Brian and Bob.
End.
```

which defines a distinct set of elements for the same class. It is the responsibility of the system builder to ensure that such behavior does not adversely affect the integrity of his code.

7.2 RELATIVE CLAUSES

```
<rel clause> ::= <disj clause>

<disj clause> ::= <conj clause> [OR <disj clause>]*

<conj clause> ::= <clause form> [AND <conj clause>]*

<clause form> ::= ( <rel clause> )
                ::= <such that/where>
                ::= <that/which/who>
                ::= <whose>
                ::= <which/whom>
                ::= <except>

<such that/where> ::= ( <st/w> <condition> )
                    ::= <st/w> <primary>

<st/w> ::= SUCH THAT
        ::= WHERE

<that/which/who> ::= <t/w/w> [<term>] <verb phrase>
                  ::= <t/w/w> <special vp>
                  ::= <t/w/w> <term> <rel op>

<t/w/w> ::= THAT
        ::= WHICH
        ::= WHO

<whose> ::= WHOSE <description> <be aux> <term>
```

`<which/whom> ::= <prep> <w/w> <term> <verb phrase>`

`<w/w> ::= WHICH
 ::= WHOM`

`<except> ::= EXCEPT <term>`

Descriptions can be modified by a relative clause, which acts like a filter on the elements of the description's class. Each time an element is generated from or compared to the elements of the description's class, the constraints specified by the relative clause are tested. If the test succeeds, the generation or comparison is allowed to continue, otherwise it is aborted.

A relative clause is actually a specialized form of *condition* (see Chapter 6), representing a boolean combination of sentences. The sentences used in a relative clause can be specified with a syntax that corresponds to English grammar. For example, the sentence,

there is a command center which is situated on hill #3

essentially represent the condition,

there is a command center and that command center is
situated on hill #3

There is a variety of forms that the sentences in a relative clause can take, many of which allow implicit reference to the element being described. Logical groups of relative clause forms can be built over conjunction and disjunction.

7.2.1 Logical Groupings

Boolean combinations of relative clause forms can be constructed with the conjunctive **and** and the disjunctive **or**. For instance, the actions

**Display every city which does support music
and whose population is small.**

**Display every club of which John is a member and
which John does attend regularly or
of which Bill is a member and
which Bill does attend regularly.**

demonstrate legal combinations of clauses.

Both the conjunctive **and** and disjunction **or** are right associative. Conjunction has a greater precedence than disjunction, and thus binds relative clause forms more tightly. The second example above would be interpreted as

Display every club (of which John is a member and
which John does attend regularly) or
(of which Bill is a member and
which Bill does attend regularly).

where parentheses designate the logical groupings of clauses. As with conditions, the default precedence of the logical connectors can be overridden with parentheses.

7.2.2 SUCH THAT... and WHERE...

```
<such that/where> ::= ( <st/w> <condition> )
                  ::= <st/w> <primary>

<st/w> ::= SUCH THAT
        ::= WHERE

<primary> ::= ( <condition> )
            ::= <sentence>
```

The most general relative clause forms are those introduced by the words **such that** and **where**. When delimited with parentheses, they can be followed by any condition, otherwise by any arbitrary sentence. This form of relative clause does *not* implicitly reference the description being modified; such a reference must appear explicitly.

Examples--

**For each integer from 1 to 100 (where that integer is even
and that integer is a multiple of 3),
display that integer.**

For each integer from 1 to 100,
if that integer is even and that integer is a multiple of 3,
display that integer.

Display every employee such that that employee does play tennis.

For each employee,
if that employee does play tennis,
display that employee.

Assert John is a man where John is happy.

Assert John is a man and assert John is happy.

In the example pairs seen above (and in similar examples seen throughout the remainder of this chapter), each action in regular font has equivalent semantics to the action in boldface preceding it.

7.2.3 THAT..., WHICH..., and WHO...

Next in generality come relative clause forms introduced by **that**, **which**, and **who**. There are two forms that these clauses can take. While both make an implicit reference to the modified description, each form differs in where it places that reference in its target relation.

First Argument Forms

```
<that/which/who> ::= <t/w/w> <verb phrase>
                  ::= <t/w/w> <special vp>
```

```
<t/w/w> ::= THAT
          ::= WHICH
          ::= WHO
```

In one form, **that**, **which**, or **who** is followed by a verb phrase (i.e., either the generalized verb phrase of a proposition or the special verb phrase of some other sentence form). Clauses of this type represent a relation with an implicit reference to the description as the subject (i.e., first argument) of the given verb phrase.

Examples--

For each integer (which is greater than 10 and
which is less than 25),
display that integer.

For each integer (where that integer > 10 and
that integer < 25),
display that integer.

Display every mother who has a son.

Display every mother where that mother has a son.

Display every employee who does play tennis.

Display every employee where that employee does play tennis.

Assert John is a man who is happy.

Assert John is a man where John is happy.

Second Argument Forms

```
<that/which/who> ::= <t/w/w> <term> <verb phrase>
                  ::= <t/w/w> <term> <rel op>
```

```
<t/w/w> ::= THAT
          ::= WHICH
          ::= WHO
```


This other form is similar to the first with the exception that the first argument of the target relation is given, and the implicit reference becomes the direct object of the relation.

Examples--

Display every diplomat who the President is meeting on Thursday.

Display every diplomat where the President is meeting
that diplomat on Thursday.

Signal every command station which the red team did not destroy.

Signal every command station where the red team did
not destroy that station.

**For each value which any result is equal to,
display that value.**

For each value where any result is equal to that value,
display that value.

7.2.4 WHOSE...

`<whose> ::= WHOSE <description> <be aux> <term>`

This relative clause form represents a class relation. The specified description is additionally modified by a prepositional phrase introduced by the preposition **of**. The object of the preposition is an implicit reference to the host description.

Examples--

Display the fleet whose flagship was ship #5.

Display the fleet where ship #5 was a flagship of that fleet.

Display every list whose member at 1 is any integer.

Display every list where any integer is a member of that list at 1.

7.2.5 WHICH... and WHOM...

`<which/whom> ::= <prep> <w/w> <term> <verb phrase>`

`<w/w> ::= WHICH
 ::= WHOM`

This relative clause form represents a proposition whose verb has been additionally modified by a prepositional phrase introduced by `<prep>`. The object of the preposition is an implicit reference to the host description.

Examples--

Display the man for whom the bell does toll.

Display the man for whom the bell does toll.

**Attack the hill on which the red army did locate
the command center.**

Attack the hill where the red army did locate
the command center on that hill.

**For each list of which any integer is a member at 1,
display that list.**

For each list where any integer is a member of that list at 1,
display that list.

**Assert AAAI is a professional organization to
which John does belong.**

Assert AAAI is a professional organization where
John does belong to that organization.

7.2.6 EXCEPT...

`<except> ::= EXCEPT <term>`

The final relative clause form provides a terse form of **is not equal to** and filters out any element of the host description that is **equal** (see Section 9.1.3) to `<term>`.

Examples--

Signal every command station except any red unit.

Signal every command station where that command station is
not equal to any red unit.

**For each member except either Jim, Jack, or John,
display that member.**

For each member where that member is not equal to either
Jim, Jack, or John,
display that member.

7.3 DESCRIPTION VARIABLES

A normally invisible yet extremely useful component of a description is its description variable. A description variable provides a unit of temporary storage. When an element is produced from a description, or tested for membership against a description, or

asserted or denied as an instance of a description, it is cached under the description variable. Such elements can be accessed later within the rule in which the description appears.

For instance, in

For each integer from 1 to 10, display that integer.

the description variable of **integer from 1 to 10** is successive bound to an integer in the range of one to ten. In each iteration, that value is accessed with the *anaphoric term*, **that integer**.

7.3.1 Anaphoric Terms and Rule Variables

A description variable can be referenced (and its value accessed) both implicitly with an *anaphoric term* and explicitly with a *rule variable*. Note that most of the relative clause forms automatically reference the description variable of the description they modify.

An anaphoric term is composed of the function word **that** preceded by the class noun of the description being referenced (e.g., **that integer**). The parser turns this into an explicit reference to the description variable. When this reference is evaluated, it will return the value cached under the description variable. If nothing is cached, it will generate the error

Unbound ANAPHORIC TERM:
THAT *class noun*

In some cases, an implicit reference is inadequate. In order to avoid otherwise ambiguous and conflicting references, the user must supply the name of the desired description variable and then reference that variable by name. This is done with the optional <desc var> syntax and a rule variable. As an example, consider the expression

**For each integer (I) from 1 to 10,
for each integer from 1 to I, display I.**

where (I) names the description variable, and every other use of I is a rule variable that references it. As with anaphoric terms, when a rule variable is evaluated at runtime, it either returns the element cached under the description variable, or, if no such element exists, it generates the error

Unbound RULE VARIABLE:
variable

7.4 ANAPHORIC DESCRIPTIONS: SUCH...

`<description> ::= SUCH <class noun> [(<desc var>)]`

`<class noun> ::= <atom>`

Additions to existing grammar rules--

`<verb phrase> ::= IS SUCH <a/an> <class noun> [(<desc var>)]`

`<special vp> ::= THERE IS SUCH <a/an> <class noun> [(<desc var>)]`

`<desc term> ::= SUCH <a/an> <class noun> [(<desc var>)]`

Sometimes a complex or verbose description must appear more than once in a rule. Rather than requiring the repetition of the full description in the second instance, ROSIE permits the use of an anaphoric description.

An anaphoric description is composed of the function word **such** followed by the class noun of the description being referenced. The parser expands this into a copy of an earlier description using the same class noun.

For example, the following two actions are semantically equivalent

**If there is an exemplary student of mathematics
who will graduate in June,
recruit (every exemplary student of mathematics
who will graduate in June) for summer employment.**

**If there is an exemplary student of mathematics
who will graduate in June,
recruit every such student for summer employment.**

They show how anaphoric descriptions can greatly enhance the readability of otherwise verbose code.

An anaphoric description functions just like a regular description. It can be used anywhere a regular description is used as well as referenced just like a regular description. Note that an anaphoric description is not an identical copy of the description it references. The copy is identical in all aspects except its description variable. As with regular descriptions, the user has the option of explicitly naming the description variable to be used in the copy. If the user does not specify the description variable, then the system generates a new "unique" variable name.

7.5 RESOLVING ANAPHORIC REFERENCES

There are two methods for anaphorically referencing a description, i.e., with anaphoric terms and with anaphoric descriptions. Such references are resolved by the parser.

In most cases, when a description is encountered by the parser, it is indexed at the front of a list of *class noun/description* pairs. When the parser encounters an anaphoric reference, it searches this list from the front looking for the first description indexed under the class noun appearing in the reference.

This does not simply result in a right-to-left scan from the reference to the target description. The precise workings of the parser can be illustrated with the following example rule:

If there is a position which does have line-of-sight to
the position of the enemy,
deploy the unit to that position.

In this rule, **that position** refers to **a position which . . .** rather than **the position of . . .**. This is due to the latter description being processed *before* the former--because it is a component of the former--and thus appearing in the list of *class noun/description* pairs *after* it.

There are two cases where a description cannot be referenced anaphorically:

- 1) When it is used to specify an *is-a* proposition, e.g.,

<34, 57> is a position of the enemy

- 2) When it is used to specify an *intentional description* (see Section 9.9), e.g.,

'the position of the enemy'

Note that in each case the user still has the option of explicitly naming the description variable used and referencing it with a rule variable. Note also that descriptions associated terms, e.g., **the enemy**, can still be referenced anaphorically.

7.6 USES OF DESCRIPTIONS

As mentioned earlier, descriptions are used by ROSIE in three ways:

- 1) testing an element for membership in the set being described

- 2) generating one, some, or all members of that set
- 3) adding or removing members from that set

7.6.1 Testing for Membership

Propositions of the form

element is a description

can be used to test *element* for membership among the set of elements named by *description*.

In performing such a test, ROSIE first determines if *element* is a member of *description*'s class, as described in Section 7.1.1. If so, then *element* is cached under *description*'s description variable, and the relative clause of *description* is evaluated. If there is no relative clause, or if it evaluates to true, then the test succeeds. Otherwise, the description variable is unbound and the test fails.

7.6.2 Generating Elements

Many constructs use descriptions to generate elements (initiating generate events), e.g.,

```
for each description . . .
there is a/an description . . .
the description . . .
every description . . .
```

Elements are generated from *description* in the following manner:

- 1) The construct establishes a halting condition,⁷ and then initiates a generate event on *descriptions*.
- 2) For each successive element that can be produced by *description*'s class (see Section 7.1.2), that element is cached under *description*'s description variable. Then, *description*'s relative clause, if any, and the halting condition are evaluated in turn. If either condition fails, then the description variable is unbound and the next element is tried. In the case where the relative clause fails, the halting condition is not evaluated.

When generate event is concluded, *description*'s description variable will be bound to the last element produced, but only if that element

⁷The halting condition depends upon the construct. For instance, a construct such as **the** will halt on the first element produced, while **every** won't halt while there is an element that can be produced.

satisfied the relative clause and the halting condition, otherwise the description variable will be unbound.

7.6.3 Asserting and Denying Members

An element can be added or removed from the set of elements named by a description in several ways. One way is to pass a proposition of the form

element is a/an description

to **assert** or **deny**, while another is to make the element the distinguished member of the set using **let**, i.e.,

let element be the description

A system generated element can be added to the set with **create**, i.e.,

create a/an description

For any of these actions, the relative clause of *description* may only contain clause forms that expand into propositions. Further, these forms may only be joined over conjunction.

While the semantics of the database actions mentioned above are described in Section 5.3 and again, more fully, in Chapter 10, the following sample session demonstrates their application.

```
(R)
[ ROSIE Version 3.0 (PSL) 26-May-86 ]

<1> Create a battalion which is on alert.
<2> ?
[ GLOBAL Database ]
    BATTALION #1 IS ON ALERT.
    BATTALION #1 IS A BLUE BATTALION.

<3> Assert battalion #1 is a blue battalion that was deployed to
      sector #12 and for which any red battalion is looking.
<4> ?
[ GLOBAL Database ]
    BATTALION #1 IS ON ALERT.
    BATTALION #1 WAS DEPLOYED TO SECTOR #12.
    ANY RED BATTALION IS LOOKING FOR BATTALION #1.
    BATTALION #1 IS A BLUE BATTALION.

<5> Deny battalion #5 is a blue battalion which was deployed
      to sector #12.
```

```

<6> ?
[ GLOBAL Database ]
  BATTALION #1 IS ON ALERT.
  ANY RED BATTALION IS LOOKING FOR BATTALION #1.

```

```

<7> Let the blue battalion whose strength is 25 be battalion #1.
<8> ?
[ GLOBAL Database ]
  BATTALION #1 IS ON ALERT.
  BATTALION #1 IS A BLUE BATTALION.
  25 IS A STRENGTH OF BATTALION #1.

```

7.7 COMPOUND CLASSES VERSUS ADJECTIVES

ROSIE 3.0 diverges from earlier ROSIEs by one significant aspect in its definition of classes and descriptions. While this change will not create any syntactic incompatibilities with existing code, it may introduce problems with code behavior. This change surrounds the notion of *compound classes* versus descriptions modified by *adjectives*. For instance,

football player

is an instance of a compound class, while

big bad burly man

is an instance of a description modified by three adjectives.

If you examine ROSIE's grammar, you will notice that it does not distinguish between compound nouns and adjectives. In order to make such a distinction, ROSIE's parsing mechanism requires special knowledge about lexical word classes such as adjectives. For reasons that will not be discussed here, providing ROSIE with the words in this class is somewhat more problematic than providing ROSIE with its knowledge of prepositions.

Earlier releases of ROSIE traditionally opted for adjectives over compound classes. This definition was considered to be more functional and flexible, the argument being that, were the distinction important, compound nouns could always be formed with hyphenation, e.g.,

football-player

Unfortunately, experience reveals that class compounds appear with great frequency in ROSIE code, and, when they appear, they must be (1) hyphenated, which detracts from code readability, or (2) emulated as descriptions modified by adjectives, which degrades system performance.

In ROSIE 3.0, the root name of a class may be one or more tokens long, thus permitting compound classes, and descriptions may only be modified by a relative clause. This eliminates the problems mentioned above, but introduces the constraint that adjective restrictions must now appear in the relative clause, e.g.,

some strategic objective

must be written as

some objective which is strategic

if **strategic** is meant to function as an adjective.

Adjectives can be emulated with *virtual relations* (see Chapter 10), e.g.,

any strategic objective is strategic

While this is not the perfect solution, it is not as awkward or inefficient as representing class compounds as descriptions with adjectives.

VIII. TERMS

This chapter introduces a basic component of any high-level language, namely the data types that the language supports and the linguistic structures that it provides for data abstraction. ROSIE's primitive data types are called *elements*, and the data abstractions used to represent elements are called *terms*. When ROSIE encounters a term during the execution of a program, it evaluates the term and passes the resulting value to the construct in which the term appeared. In this chapter, we focus on the available term forms and how these forms are evaluated. While we include a short section on elements, Chapter 9 deals with elements in greater detail.

8.1 TYPES OF TERMS

```
<term> ::= <element>
        ::= <arith expr>
        ::= <desc term>
        ::= <anaphor>
        ::= <iter term>
```

A term is a form of data abstraction representing one or possibly any number of data objects. These data objects are drawn from ROSIE's ten data primitives called *elements*. Terms serve as arguments to actions and sentences as well as other terms.

There are essentially five types of terms:

- 1) *Elements*, which evaluate to themselves
- 2) *Arithmetic expressions*, which provide a small set of infix arithmetic operations
- 3) *Descriptive terms*, which apply operators in the form of articles (e.g., **a** and **the**) and quantifiers (e.g., **some** and **every**) to descriptions and evaluate to one, some, or all of the elements described
- 4) *Anaphoric terms* and *rule variables*, which evaluate to elements generated from a description or created by the string pattern matcher
- 5) *Iterative terms*, which permit the specification of an explicit set of elements over which a test or action is iterated.

Terms can be nested to an arbitrary depth, and an arbitrary number of embedded terms can appear at the same level, e.g.,

the man (from the city (by the river (to the east
(in the valley))
(in Ohio))
(in the suit (from the store (in Chicago)))
(by the drugstore (on the corner)))

Embedded terms are normally evaluated in postorder (i.e., from left to right and from the inside out). In the above example, **the east** would be evaluated first, then **the valley**, then **the river to the east in the valley**, then **Ohio**, then **the city by the river . . .**, etc.

NOTE: While ROSIE attempts to preserve ordering whenever possible during the evaluation of terms, the definition of the language does not guarantee that ordering will always be maintained. Unless the definition of an operation clearly specifies the treatment of its arguments, users should avoid writing code that depends upon order of evaluation for proper performance.

8.2 ELEMENTS

```

<element> ::= <name element>
           ::= <number element>
           ::= <tuple element>
           ::= <string element>
           ::= <pattern element>
           ::= <filesegment>
           ::= <class element>
           ::= <intentional description>
           ::= <intentional proposition>
           ::= <intentional procedure>

```

Whenever a term is encountered as an argument to an action, or a sentence, or another term, it is evaluated. It will evaluate to one or possibly a sequence of elements, which will be passed on to the construct in which the term appears. Hence, elements are passed as the arguments of rulesets, and they are produced as the results of a generator ruleset. They appear as the arguments propositions affirmed in the database as well as the object of affirmed class relations.

Elements can be divided between two categories: *simple elements* and *intentional elements*. These include:

Simple Elements	Examples
<i>names</i>	battalion #5
<i>numbers</i>	
<i>simple numbers</i>	3.1412
<i>unit constants</i>	55 miles/hour
<i>labeled constants</i>	probability 0.75
<i>strings</i>	"The ratio HEP/COG:"
<i>patterns</i>	{{"Yes" "No"} (bind to the reply), cr}
<i>tuples</i>	<pol soft, <5 waves, FX-4>>
<i>filesegments</i>	'file: "intel", to report a finding'
 Intentional Elements	
<i>class elements</i>	any non-offensive target
<i>intentional descriptions</i>	'an action at the current time'
<i>intentional propositions</i>	'visibility does approximate 3.5 miles'
<i>intentional procedures</i>	'deploy the unit to sector #3'

Most of the simple elements correspond to the basic data types found in other symbolic languages. Elements such as *patterns* and *filesegments* interface to facilities that are unique to ROSIE. The intentional elements provide ROSIE with limited "self-referential" capabilities, allowing programs to treat units of descriptive, declarative, and procedural knowledge as data. Elements are discussed further in Chapter 9.

8.3 ARITHMETIC EXPRESSIONS

<arith expr> ::= <term> <op> <term>

**<op> ::= +
 ::= -
 ::= *
 ::= /
 ::= ^
 ::= ****

Arithmetic expressions support a predefined set of infix arithmetic operators. Arithmetic expressions are treated as terms that evaluate to number elements. Each operator is a function of two arguments, which must evaluate to number elements whose units or labels are compatible under the given operation.

8.3.1 Operators and Operations

ROSIE supports five arithmetic operators: (+) for addition; (-) for subtraction; (*) for multiplication; (/) for division; and (^) and (**) for exponentiation.¹ Note that in order for these operators to be

¹The uparrow (^) is being introduced for the first time in ROSIE 3.0.

recognized by the parser they *must* be surrounded by separator characters, e.g., `3 + 4` as opposed to `3*4`--the former will be treated as an arithmetic expression, while the latter as a name element.

The label and units of the numbers to which these operators are applied must be compatible under the following rules:

- When numbers are added (+) or subtracted (-), both numbers must be of the same type and have the same units or label. The result is a number with the same units or label.
- When numbers are multiplied (*) or divided (/), they must either both be label constants with the same label or both be unit constants with comparable units (see Section 9.3), or one of the two numbers must be a simple number. The result is a number that has the appropriate label or units.
- Exponentiation (^ and **) requires that the exponent (the second operand) be a simple number with an integer value. The other operand can be any number type. If it is a label constant, the result retains the label. If it is a unit constant, the units will also reflect exponentiation.

If the above constraints are not met, then the error message

Illegal arguments:
element op element

will be generated.

Note that these operations do not distinguish computations between real numbers and integers. If applied to two integers, an operation will attempt to return an integer, otherwise a real number is returned, e.g., `10 / 5` returns 2 while `10 / 4` returns 2.5. When at least one of the operands is a real number, the result will be a real.

As an example, consider the following expressions and their resulting values:

Example	Results
<code>3 + 4</code>	7
<code>5 apples - 2 apples</code>	3 apples
<code>time 9 + time 1</code>	time 10
<code>3 / 4</code>	0.75
<code>55 miles/hour * 3 hour</code>	165 miles
<code>time 5 * 2</code>	time 10
<code>5 ^ 2</code>	25
<code>5 ^ -2</code>	0.04
<code>3 feet ^ 3</code>	27 feet^3

Notice that ROSIE does not understand plurality in units; thus the use of **3 hour** as opposed to **3 hours**. Also note that while the arithmetic operators must be delimited by separator characters when used in arithmetic expressions, the opposite is true when used in the units of a number, e.g., **3 feet ^ 3** as opposed to **27 feet^3**.

8.3.2 Associativity and Precedence

The rules of associativity and precedence for the arithmetic operators are fairly standard. The operations of addition (+), subtraction (-), multiplication (*), and division (/) are all left associative. Exponentiation (^ and **) is right associative. Exponentiation has the highest precedence, then multiplication and division, then addition and subtraction. Thus, the expression

$$3 + 4 * 5 ^ 6 / 6 ^ 5 * 4 + 3$$

would be interpreted as

$$((3 + (((4 * (5 ^ 6)) / (6 ^ 5)) * 4)) + 3)$$

When used within a prepositional phrase, e.g.,

the absolute value of 5 * -6

precedence of the arithmetic operators is less than that of prepositions. Thus, the above example would be interpreted as

$$(\text{the absolute value of } 5) * -6$$

and not

$$\text{the absolute value of } (5 * -6)$$

8.4 DESCRIPTIVE TERMS

```
<desc terms> ::= THE <description>
               ::= <term> ' S <description>
               ::= <a/an> <description>
               ::= A NEW <description>
               ::= SOME <description>
               ::= EVERY <description>
```

Descriptive terms reference and evaluate to the elements named by a description (see Chapter 7). Descriptive terms fall into two categories, the *simple descriptive terms* and the *quantified descriptive terms*. Simple descriptive terms evaluate to one and only one of the elements named by a description, while quantified descriptive terms evaluate to a sequence of some or all of these elements.

A descriptive term consists of two components, a function word and a description. The function word can either be one of the articles, **the**, **a** or **an**, e.g.,

the *sortie rate of airfield #3*
an *emergency*

the special article form, **a new**, e.g.,

a new *rule which does conclude the hypothesis*

or one of the quantifiers, **some** or **every**, e.g.,

some *ship which is not seaworthy*
every *strike force in sector #5*

The function word serves to introduce the term and specifies how the term is to be evaluated.

When evaluated, a descriptive term initiates a generate event on its description after establishing a halting condition based upon the semantics of its function word. The halting condition is successively applied to each element named by the description until it evaluates to true, terminating the generate event. If the halting condition is successfully met, then the description variable of the term's description will be bound to the element that satisfied it. This element is treated as the value of the term and can be accessed by an *anaphoric term* or a *rule variable* (see Section 8.5).

8.4.1 Simple Descriptive Terms

Simple descriptive terms are introduced by the articles, **the**, **a** and **an**, and by the special article form, **a new**. This type of descriptive term will evaluate to one and only one element.

8.4.1.1 THE...

If the descriptive term is introduced by **the**, it will return as its value the first element that can be produced by its description. For example, consider the following sample session.

```
(R)
[ ROSIE      Version 3.0 (PSL)  26-May-86 ]

<1> Assert each of John, Jack and Joe is a man.
<2> Assert John does love Mary.
<3> ?
[ GLOBAL Database ]
  MARY IS A WOMAN.
  JOE IS A MAN.
  JACK IS A MAN.
  JOHN IS A MAN.
```

```

<4> Display the man.
JOE
<5> Display the man who does love Mary.
JOHN

```

If no element can be produced, an error is called, i.e.,

```

No such element exists:
THE description

```

This type of term is most commonly used as a variable under which the intermediate values of computation are stored. For instance, in the ruleset

```

To generate the length of a tuple:
Private: a counter.
[1] Let the counter be 0.
[2] For each member of the tuple,
    let the counter be the counter + 1.
[3] Produce the counter.
End.

```

the counter is used as a local variable to keep track of the number of elements seen in a given tuple. This example also shows the common use of the **let** database action as an assignment operator.

ROSIE permits a shorthand syntax for descriptive terms introduced by **the** via the possessive case of apostrophe *s*, e.g.,

John's mother

expands to

the mother of John

The syntax used above is *term's description*, where the parser adds **of term** to the prepositions associated with *description*, e.g.,

the city's mayor in 1971

expands to

the mayor of the city in 1971

8.4.1.2 A... and AN...

The semantics of descriptive terms introduced by the articles, **a** and **an**, are similar to **the**, with one exception. Like **the** terms, the description is requested to produce an element that becomes the value of the term. However, if no such element can be produced, an element will be created as with the **create** database action. This element will be asserted as an instance of the description and returned as the value of the term.

(R)
[ROSIE Version 3.0 (PSL) 26-May-86]

```
<2> ?
[ GLOBAL Database ]

<3> Display a truck.
TRUCK #1
<4> ?
[ GLOBAL Database ]
    TRUCK #1 IS A TRUCK.

<5> Display a truck.
TRUCK #1
```

To create an element, ROSIE takes the class noun of the description and appends the suffix **#N**, where **N**, a positive integer associated with the class noun, is incremented by one for each element so created.

8.4.1.3 A NEW...

A descriptive term introduced by the special article form **a new** *never* attempts to generate an instance of its description. Rather, it automatically creates an element, asserting that element as an instance of the description and returning that element as its value.

```
<6> Assert Bill does own a new truck and
      John does own a new truck.
<7> ?
[ GLOBAL Database ]
    JOHN DOES OWN TRUCK #3.
    BILL DOES OWN TRUCK #2.
    TRUCK #3 IS A TRUCK.
    TRUCK #2 IS A TRUCK.
    TRUCK #1 IS A TRUCK.
```

8.4.2 Quantified Descriptive Terms

Quantified descriptive terms are introduced by the quantifiers, **some** and **every**, e.g.,

some target which does satisfy every engagement requirement

providing an implicit form of iteration over the elements named by a description. Unlike the other terms we have seen, quantified descriptive terms do *not* evaluate to a single element. At a conceptual level, they evaluate to a sequence of elements. In reality, quantified descriptive terms change the structure of the expression in which they appear, causing repeated evaluation of that expression.

When an action or sentence contains a quantified descriptive term as an argument, the parser performs a transformation on that action or sentence such that it is embedded in an iterative loop. The loop operator takes two arguments, i.e., (1) the description and (2) the expression over which to iterate. The occurrence of the quantified term in the expression is replaced with a reference to its description variable. Thus, the semantics of an action such as

Display every integer from 1 to 10.

are equivalent to

For each integer from 1 to 10, display that integer.

The loop operator actually works by initiating a generate event on the given description. The halting condition for this event contains the transformed expression, which can be evaluated repeatedly as each element is produced. In addition, the halting condition also contains a cue that decides when to terminate the generate event. The semantics of this cue depends upon the quantifier and the type of expression in which the term appears.

When the expression in question is an action, the iterative loop simply executes the action repeatedly. When the expression is a sentence used in a test, e.g.,

If some target is vulnerable . . .

then the loop repeatedly evaluates the sentence until it tests true or false depending on the quantifier. For instance, the above example is equivalent to

If there is a target where that target is vulnerable . . .

Note that a quantified descriptive term only changes the immediate sentence or action in which it appears. If it is nested within a term to any depth, then it changes the immediate sentence or action in which that term appears, e.g., the action,

Display the name of every man.

is equivalent to

For each man, display the name of that man.

Note also that when a quantified term appears within a relative clause, e.g.,

Display the target which does satisfy every requirement.

then transformation is applied to that relative clause form (because it is a sentence). Hence, the above example is equivalent to²

Display the target such that there is no requirement which
that target does not satisfy.

8.4.2.1 SOME...

The quantifier **some** causes iteration to continue until it finds one element that satisfies the expression in which the quantified term appears. Since an action can be said to be ambivalent to any argument it receives, this type of term is more appropriate to use within a sentence. An action that contains a **some** term will be executed only once, using the first element produced from the term's description. A sentence will be tested repeatedly on each element until the test succeeds.

For instance, the action,

Dispatch some fighter to the target.

will be transformed by the parser into

If there is a fighter,
dispatch that fighter to the target.

while the sentence,

some fighter is unassigned

²Actually, a precise paraphrase would be

Display the target such that there is no requirement
where 'that target does satisfy that
requirement' is not provably true.

However, intentional propositions and the **is provably** predicate have not yet been introduced, and, since they tend to otherwise obscure the examples in this section, they are not used here.

from the iterative action,

*While some fighter is unassigned,
dispatch that fighter to the target.*

will be transformed into

*While there is a fighter which is unassigned,
dispatch that fighter to the target.*

For more elaborate applications, consider the following actions,

*For each fighter which is assigned to some target,
display that fighter.*

*For each target at some airfield which is on alert,
defend that target.*

and their equivalents without quantifiers,

*For each fighter such that there is a target
where that fighter is assigned to that target,
display that fighter.*

*If there is an airfield which is on alert,
for each target at that airfield, defend that target.*

The first example demonstrates the interpretation of a **some** term that appears in a relative clause. This can be compared to the second example in which the **some** term is a component (i.e., object of a preposition) of a component (i.e., the description being iterated over) of an iterative action.

8.4.2.2 EVERY...

The **every** quantifier is essentially the inverse of **some**, causing iteration to continue until it finds one element that fails to satisfy the expression in which the quantified term appears. An action that contains an **every** term will be executed for each element produced. A sentence will be tested until an element fails to satisfy the test or until all elements have been produced, in which case the test succeeds.

The action,

Dispatch every fighter to the target.

will be transformed by the parser into the equivalent of

For each fighter, dispatch that fighter to the target.

and the sentence,

every fighter is assigned

from the iterative action,

If every fighter is assigned, initiate the strike.

will be transformed into the equivalent of

*Unless there is a fighter which is not assigned,
initiate the strike.*

For more elaborate applications, consider the following example pairs:

**For each fighter which did strike every objective,
display that fighter.**

**For each target at every airfield which is on alert,
defend that target.**

and their nonquantified equivalents:

For each fighter such that there is no objective
which that fighter did not strike,
display that fighter.

For each airfield which is on alert,
for each target at that airfield, defend that target.

The first example demonstrates the interpretation of an **every** term that appears in a relative clause. This can be compared to the second example in which the **every** term is a component (i.e., object of a preposition) of a component (i.e., the description being iterated over) of an iterative action.

8.5 ANAPHORIC TERMS AND RULE VARIABLES

`<anaphor> ::= THAT <class noun>
 ::= <rule var>`

`<rule var> ::= <desc var>`

Anaphoric terms and rule variables provide a means of referencing elements produced from a description. They do this by making either an implicit (as in the case of anaphoric terms) or explicit (as with rule variables) reference to the description variable associated with the target description. At runtime, such a reference evaluates to the element stored under that description variable.

An anaphoric term is composed of the function word **that** preceded by a class noun of some description appearing earlier in the rule in which the term appears.³ For example, in

For each positive integer from 1 to 10, display that integer.

that integer references the description **positive integer from 1 to 10**.

When encountered at parse time, the parser expands an anaphoric term into an explicit reference to the description variable of the target description. When this reference is evaluated, it will return the valued cached under the description variable. If nothing is cached, it will generate the error,

Unbound ANAPHORIC TERM:
THAT *class noun*

By supplying the name of the description variable the user can make explicit reference to it by using a rule variable of the same name, e.g.,

For each positive integer (I) from 1 to 10, display I.

When the parser encounters a single word name, such as **I**, which is the same as an explicitly designated description variable, it treats the name as a rule variable and translates it into a explicit reference to the description variable. As with anaphoric terms, when this reference is evaluated, it either returns the element cached under the description variable, or, if no such element exists, it generates the error,

Unbound RULE VARIABLE:
variable

The scope of anaphoric terms and rule variables is limited to the rule in which the target description is evaluated. A description variable cannot be referenced, nor its value accessible outside of this rule.

8.6 ITERATIVE TERMS

```
<iter term> ::= ONE OF <term> [, <term>]* [, ] OR <term>
              ::= EITHER <term> [, <term>]* [, ] OR <term>
              ::= EACH OF <term> [, <term>]* [, ] AND <term>
              ::= BOTH <term> [, ] AND <term>
```

³For further details, see Section 7.5.

The iterative terms provide a unique device for looping over a group of elements to perform an action or test a condition. They are essentially a specialized form of the quantified descriptive terms in which the elements over which iteration is to be performed are explicitly named.⁴ Like quantified descriptions, the iterative terms actually change the structure of the action or sentence in which they appear.

There are two types of iterative terms, the *disjunctive iterators* and the *conjunctive iterators*. Both are characterized by a term list, where each term is separated by a comma (,) and the last term is separated from the others by the disjunctive **or** or the conjunctive **and**, respectively. When iterating over the term list, successive terms are evaluated one by one as needed.

The disjunctive iterators are introduced by the phrase **one of** or **either**, corresponding to the descriptive terms introduced by the quantifier **some**. A conjunctive iterator must be introduced by the phrase **each of**, which corresponds to the **every** quantified descriptive term. A conjunctive form of just two elements can be introduced by the word **both**.

8.6.1 ONE OF... and EITHER...

The **one of** and **either** disjunctive iterators will loop over an expression for each of the elements from their term list until one of these elements satisfies some halting criteria. An action that contains such an iterative term will be executed only once, using the first element from the term list.⁵ A sentence containing a disjunctive iterator will be tested for each element from the term list until the test succeeds.

For instance, the sentence,

either fighter #1, fighter #2 or fighter #3 is unassigned

from the iterative action,

*While either fighter #1, fighter #2 or fighter #3 is unassigned,
dispatch a fighter to the target.*

will first test the '*element is unassigned*' predicate with **fighter #1** as *element*. If that test succeeds, then the **dispatch** procedure is executed, otherwise the test is applied to **fighter #2**, etc.

⁴In the old manual, the quantified descriptive terms and the iterative terms were grouped together in a single class called the *pseudo-terms*.

⁵While this semantic is recognized as not being an especially useful one, it is nonetheless included for completeness.

8.6.2 EACH OF... and BOTH...

Like **one of** and **either**, the iterative terms introduced by **each of** and **both** loop over an expression for each of the elements from their term list until one of these elements satisfies some halting criteria. This criterion, however, is the inverse of that used by the former. An action that contains a conjunctive iterator will be executed for each element from the term list. A sentence will be tested repeatedly, using each element until one of the elements causes the test to fail or the term list is exhausted, in which case the test succeeds.

For instance, the action,

Dispatch each of fighter #1, fighter #2 and fighter #3.

will apply the **dispatch** procedure in turn to **fighter #1**, **fighter #2**, and **fighter #3**, while the sentence,

both fighter #1 and fighter #2 is assigned

from the iterative action,

*If both fighter #1 and fighter #2 is assigned,
initiate the strike.*

will succeed if '**fighter #1 is assigned**' and then '**fighter #2 is assigned**' both test true.

IX. ELEMENTS

This chapter reintroduces *elements*, ROSIE's data primitives. ROSIE's elements define its space of concepts. These elements include *the simple elements: names, numbers, strings, patterns, tuples, and filesegments*; and *the intentional elements: class elements, intentional descriptions, intentional propositions, and intentional procedures*. The simple elements consist of variations on the basic data types found in most symbolic programming languages, while the intentional elements provide a means of treating units of descriptive, declarative, and procedural knowledge as data.

9.1 ELEMENT BASICS

```
<element> ::= <name element>
           ::= <number element>
           ::= <tuple element>
           ::= <string element>
           ::= <pattern element>
           ::= <filesegment>
           ::= <class element>
           ::= <intentional description>
           ::= <intentional proposition>
           ::= <intentional procedure>
```

ROSIE's primitive data types are called *elements*. All terms evaluate to elements. Rulesets are passed elements as arguments and can be defined to generate a sequence of elements on request. Elements can be stored as the arguments of declarative relations and affirmed as instances of class relations.

9.1.1 Types of Elements

Elements can be divided between two categories: *simple elements* and *intentional elements*. These include:

Simple Elements	Examples
<i>names</i>	battalion #5
<i>numbers</i>	
<i>simple numbers</i>	3.1412
<i>unit constants</i>	55 miles/hour
<i>labeled constants</i>	probability 0.75
<i>strings</i>	"The ratio HEP/COG:"
<i>patterns</i>	{{"Yes" "No"}} (bind to the reply), cr}
<i>tuples</i>	<pol soft, <5 waves, FX-4>>
<i>filesegments</i>	'file: "intel", to report a finding'

Intentional Elements

<i>class elements</i>	any non-offensive target
<i>intentional descriptions</i>	'an action at the current time'
<i>intentional propositions</i>	'visibility does approximate 3.5 miles'
<i>intentional procedures</i>	'deploy the unit to sector #3'

Several of the simple elements (i.e., *names*, *numbers*, *strings*, and *tuples*) exist as slightly more complex variations on the basic data types found in most symbolic programming languages. The other simple elements provide explicit representation for data structures used in operations that are unique to ROSIE. For instances, *filesegments* identify portions of a program file that can be manipulated via the file package, and *patterns* interface to ROSIE's string pattern matcher and support complex input and output operations.

The intentional elements provide ROSIE with limited "self-referential" capabilities, allowing programs to treat units of descriptive, declarative, and procedural knowledge as data. *Class elements* and *intentional descriptions* permit program control over the retrieval and definition of class relations, e.g.,

Execute every instance of 'an action at the current time'.

Intentional propositions capture the intent of relations between objects, which can be passed as arguments to rulesets, e.g.,

Report 'visibility does approximate 3.5 miles'.

as well as asserted, tested, or denied. *Intentional procedures* provide a representation for working with suspended actions, e.g.,

Execute 'deploy the unit to sector #3' at time 100.

which can be queued and later executed on demand. Essentially, the intentional elements give knowledge engineers a vehicle for developing meta-level control mechanisms.

9.1.2 Evaluation Names

Every element has an *evaluation name*, which is its character string representation. Whenever an element is sent to an output device, such as the user's terminal, or coerced into a string element, its evaluation name is used. Each type of element has its own format for creating an evaluation name such that, if parsed and evaluated, it would return the original element.

For example, consider the following term,

<the general, 3 + 4, John's mother, John>

which is a *tuple element* with four embedded terms: **the general**, a descriptive term; **3 + 4**, an arithmetic expression; **John's mother**, shorthand for **the mother of John**; and **John**, a *name element*. Assuming the embedded terms evaluate to **George Custard**, **7**, **Sara Lee**, and **John**, respectively, the tuple would evaluate to

<GENERAL CUSTARD, 7, SARA LEE, JOHN>

This would likewise become the evaluation name of the tuple and would appear whenever the tuple was sent to an output device or coerced into a string.

9.1.3 Equivalence versus Equality

ROSIE supports two comparison operations for determining the "sameness" of elements, *equality* and *equivalence*. Each is an operation of two arguments, defined as follows:

- *Equality* succeeds if its arguments are of the same type and produce the same evaluation name.¹
- *Equivalence* succeeds if one or both arguments can be coerced into equal elements.

According to the above, equality is defined as one might think, while equivalence is essentially defined as a test of set intersection. This is an important distinction to remember, because the "equality" special sentence forms, i.e.,

<term> is [not] equal to <term>
<term> [~]= <term>

actually test equivalence. Whenever we talk about comparing two elements for equality using the above operators, we are really talking

¹Assume the "sameness" of evaluation names is tested with the LISP **equal** function.

about equivalence. Our earlier definition of "strict" equality is only supported for internal operations and is *not* accessible to the user.

In practice, the only situation in which equality and equivalence produce distinctly different behavior occurs when one or both arguments are *class elements* (i.e., a description introduced by the function word *any*), or when a class element is nested within either argument. A class element functions as a "wild card" element, matching any instance of its designated class. Thus, the test

John = any man

will succeed only if some instance of **man** is equivalent to **John**. Such cases as

any mortal = any man

will succeed only if some instance of **mortal** is equivalent to some instance of **man**.

To be more precise, equivalence is a function of two arguments described recursively as follows:

- 1) If both arguments are class elements, then they are equivalent if some instance of the second argument is provably an instance of the first, e.g.,

any mortal = any man

only if

some man is a mortal

- 2) If only one argument is a class element, then they are equivalent if the argument that is not a class element is provably an instance of the class, e.g.,

John = any man

only if

John is a man

- 3) If both arguments are of the same type and contain *no* embedded elements (i.e., names, numbers, and strings), then they are equivalent if they produce the same evaluation name.
- 4) Finally, if both arguments are of the same type but *do* contain embedded elements (i.e., tuples, patterns, filesegments, and the intentional elements), then the equivalence test is applied recursively to the corresponding elements of each. The two arguments are equivalent if their fixed parts are the same and their corresponding elements are equivalent, e.g.,

$\langle \text{any man, Mary} \rangle = \langle \text{John, any woman} \rangle$

only if

John = any man

and

Mary = any woman

Additional constraints exist on elements containing descriptions (i.e., intentional descriptions, and intentional propositions that specify class relations) when their descriptions are modified by a relative clause. For instance, the intentional proposition

'John is a salesman from Detroit who does sell shoes in Baltimore'

contains a description modified by the relative clause

who does sell shoes in Baltimore

In such cases, the "embedded" elements include the subject, the direct object (if any), and the objects of prepositions (e.g., **John** and **Detroit**) but not the terms within the relative clause (e.g., **Baltimore**). In order for two corresponding descriptions to be equivalent, they *must* use the same sequence of tokens in their relative clauses, e.g.,

'the man who is happy'

is equivalent to

'the man who is happy'

but not

'the man where that man is happy'

That is to say, ROSIE does not test for logical equivalence of relative clauses.

9.1.4 General Operations on Elements

In the following operations *an element* can be any type of element.

an element is a thing

Always concludes true, i.e., every *element* is a member of the class **thing**.

The **thing** class, when used in a class element, is actually more functional than a cursory glance would suggest. For instance, say you wanted to define the **vessel** class as any element that is the subject of a **does float** relation, regardless of whether it is a

ship, boat, dinghy, raft, etc. This can be done simply by asserting the proposition,

any thing that does float is a vessel

Then, testing '*element is a vessel*' would first test '*element is a thing*' (always succeeding) and then test '*element does float*'.

Note however that one may not generate from the **thing** class, i.e., evaluating the term,

every vessel

will *not* produce all subjects of affirmed **does float** relations.

an element is a name

Concludes true if *element* is a name element, false otherwise.

an element is a number

Concludes true if *element* is a number element, false otherwise.

an element is a tuple

Concludes true if *element* is a tuple element, false otherwise.

an element is a string

Concludes true if *element* is a string element, false otherwise.

an element is a pattern

Concludes true if *element* is a pattern element, false otherwise.

an element is a filesegment

Concludes true if *element* is a filesegment, false otherwise.

an element is a class

Concludes true if *element* is a class element, false otherwise.

an element is a description

Concludes true if *element* is an intentional description, false otherwise.

an element is a proposition

Concludes true if *element* is an intentional proposition, false otherwise.

an element is a procedure

Concludes true if *element* is an intentional procedure, false otherwise.

the element type of an element

Produces the type of *element*, i.e., one of

NAME	FILESEGMENT
NUMBER	CLASS
TUPLE	DESCRIPTION
STRING	PROPOSITION
PATTERN	PROCEDURE

an element [~]= an element***an element is [not] equal to an element***

Concludes true if both *elements* are equivalent, false otherwise. Element equivalence is defined in Section 9.1.3.

an instance of an element

If *element* is an intentional description or a class element, produces successive instance of its class, otherwise simply produces *element*, e.g.,

```
<6> Assert each of Jim, Jack and Joe is a man.
<7> Display every instance of 'a man'.
JOE
JACK
JIM
<8> Display every instance of a man.
JOE
```

an argument of an element

If *element* takes other elements as arguments (i.e., tuples, patterns, filesegments, and the intentional elements), successively produces those elements, otherwise produces nothing, e.g.,

```
<9> Display every argument of <A, B, C>.
A
B
C
<10> Display every argument of 'John does love Mary'.
JOHN
MARY
```

the substitution of an element for an element in an element

Produces a copy of the third *element* with the first occurrence of the second *element* replaced with the first *element*, e.g.,

```
<1> Display the substitution of Bill for John in 'John is a man'.
'BILL IS A MAN'
```

substitute an element for an element in an element

Destructively substitutes the first *element* for the first instance of the second *element* in the third *element*, e.g.,

```
<5> Let the sentence be 'John is a man'.
<6> ?
[ GLOBAL Database ]
  'JOHN IS A MAN' IS A SENTENCE.

<7> Substitute Bill for John in the sentence.
<8> ?
[ GLOBAL Database ]
  'BILL IS A MAN' IS A SENTENCE.
```

a copy of an element

Produces a copy of *element*, e.g., given the previous example

```
<9> Let the test sentence be a copy of the sentence.
<10> ?
[ GLOBAL Database ]
  'BILL IS A MAN' IS A TEST SENTENCE.
  'BILL IS A MAN' IS A SENTENCE.

<11> Substitute Joe for Bill in the test sentence.
```


<12> ?

[GLOBAL Database]

'JOE IS A MAN' IS A TEST SENTENCE.

'BILL IS A MAN' IS A SENTENCE.

9.2 NAMES

`<name element> ::= [<atom>]* <atom>`

A name element is a sequence of one or more nonreserved tokens. Names are most often employed as labels on abstract concepts. Names can also be used as abbreviations for identify *filesegment elements* (see Section 9.7) as well as *alternate databases* (see Chapter 10).

Examples of legal names--

John
Mr. John Smith
General George Custard
Battalion #5
PA 6-5000
Washington State University

Names cannot include strings or numbers.

Examples of illegal names--

John "The Smasher" Brown
Employee 0029 A4
I.R.S. Section 319

Note that while the first two examples will cause a parsing error, the third will not; **I.R.S. Section 319** will be interpreted as a number (i.e., a *labeled constant*).

9.3 NUMBERS

```

<number element> ::= <simple number>
                  ::= <unit constant>
                  ::= <labeled constant>

<simple number> ::= <number>

<unit constant> ::= <number> [<atom>]* <atom>

<labeled constant> ::= [<atom>]* <atom> <number>

<number> ::= <integer>
           ::= <real>

<integer> ::= [+/-]nnn

<real> ::= [+/-]nnn.[nnn][E[+/-]nnn]
         ::= [+/-][nnn].nnn[E[+/-]nnn]
         ::= [+/-]nnnE[+/-]nnn

```

A number element is used to represent numeric values. Numbers can be of three types: *simple numbers*, *unit constants*, and *labeled constants*. ROSIE's arithmetic operators (+ - / * ^), as well as its comparison operators (= > < >= <=) and their complements (~= ~> ~< ~>= ~<=), combine and compare units and labels in a manner which should seem intuitively correct. They also ensure that these combinations and comparisons are sensible. For example, attempting to add **3 apples** to **2 oranges** will cause a mismatched units error.

Units and labels improve the expressiveness and readability of numeric computations. They greatly enhance the representational power of numbers, making their occurrence in code more meaningful.

9.3.1 Types of Numbers

There are three types of number elements: *simple number*, *unit constants*, and *labeled constants*. These are defined below.

Simple Numbers

A simple number can be expressed as either an integer or a real, e.g.,

```

10
2.718

```

with no associated units or labels. By this definition, simple numbers subsume the concept of numbers found in most programming languages.

Unit Constants

A *unit constant* consists of an integer or a real followed by one or more nonreserved atomic tokens, e.g.,

```
3 apples
55 miles/hour
20.8 feet*pounds/seconds^2
200 metric tons/cubic feet
13.7 1/feet^2
13.7 feet^-2
```

These tokens represent composite units of measure that can be combined under multiplication, division, or exponentiation.² A precise definition of the syntax of unit constants is demonstrated by the following BNF

```
<unit constant> ::= <number> [1/]<units>

<units> ::= <unit>
          ::= <unit>*<units>
          ::= <unit>/<units>
          ::= <unit>^<integer>
          ::= <unit>**<integer>

<unit> ::= [<atom>]* <atom>
```

where (*) and (/) represent multiplication and division, and (^) and (**) both represent exponentiation. The (*) and (/) operators have equal precedence and are left associative; the (^) and (**) operators have a higher precedence and are right associative.

ROSIE's basic arithmetic operators know how to combine units, e.g., the unit constant

4 R*S/T*U^2/V

can be created by evaluating the arithmetic expression

4 * (((1 R) * (1 S)) / ((1 T * (1U ^ 2)) / 1 V))

When creating an evaluation name for a unit constant, ROSIE always flips units with negative exponents in the denominator. Thus,

4 R*S*V/T*U^2

²Except for the ability to recognize and combine embedded operators, ROSIE does not have any special knowledge concerning units of measure.

would be the evaluation name of the example number illustrated above.

Labeled Constants

A labeled constant consists of an integer or a real preceded by one or more nonreserved tokens referred to as its *label*, e.g.,

certainty 0.75
Ground Combat Division 13

9.3.2 Constraints on Numbers

The following constraints apply to how numbers may be specified, compared, and combined:

- There is no such thing as a unit constant with a label or a labeled constant with units. Thus, trying to specify a number such as

probability 75 percent

will result in the parser generating a syntax error.

- When numbers are compared using the operations ($>$ $<$ $>=$ $<=$) or their complements ($\sim>$ $\sim<$ $\sim>=$ $\sim<=$), both numbers must be of the same type and have the same units or label; if not, an error occurs.
- When numbers are added (+) or subtracted (-), both numbers must be the same type with the same units or labels.
- When numbers are multiplied (*) or divided (/), they must both be labeled constants with identical labels, or unit constants (units do *not* have to be identical), or one of the numbers must be a simple number.
- The exponentiation operators (^) and (**) require that the exponent be a simple number with an integer value; the other operand can be a number of any type.

9.3.3 Operations on Numbers

In the following operations, *a number* refers to a number element, *an integer* refers to a simple number with an integer value, and *an element* refers to an arbitrary element.

an element is a number
an element is a positive number
an element is a negative number
an element is a simple number

an element is an integer
an element is a positive integer
an element is a negative integer
an element is a unit constant
an element is a labeled constant

The above predicates conclude true if *element* is a number element with the prescribed properties.

NOTE: The integer properties refer to simple numbers only.

the units of a number

Produces a name element representing the units of *number*.

the label of a number

Produces a name element representing the label of *number*.

a number + a number
a number - a number
*a number * a number*
a number / a number
a number ^ an integer
*a number ** an integer*

ROSIE provides the above infix arithmetic operators:

precedence	associativity	operation
\wedge **	right	(exponentiation)
* /	left	(multiplication and division)
+ -	left	(addition and subtraction)

The label and units of the numbers to which these operators are applied must be compatible according to the following rules:

- When numbers are added (+) or subtracted (-), both numbers must be of the same type and have the same units or labels. The result is a number of the same type with the same units or labels.
- When numbers are multiplied (*) or divided (/), they must either both be labeled constants with the same label or both be unit constants with comparable units, or one of the two numbers must be a simple number. The result is a number that has the appropriate label or units.

- The exponent operation (^) and (**) requires that the exponent (i.e., the second argument) be a simple number with an integer value. The other operand can be any number type. If it is a labeled constant, the result retains the same label. If it is a unit constant, the units also will reflect exponentiation.

If the above constraints are not met, then the error message

Illegal arguments:
element op element

will be generated.

a number [~]>[=] a number
a number is [not] greater than [or equal to] a number

a number [~]<[=] a number
a number is [not] less than [or equal to] a number

The following special sentence forms are available for comparing number elements and have the obvious results. Comparisons can be made only between numbers with the same units or labels, otherwise an error occurs.

a number does range from a lower bound to an upper bound

Concludes true if *number* is greater than or equal to *lower bound* and less than or equal to *upper bound*.

the numeric value of a number

Produces a simple number representing the value of *number*.

the absolute value of a number

Produces the absolute value of *number*, preserving units or labels.

the negation of a number

Produces the negation of *number*, preserving units or labels.

the floor of a number

Produces the floor of *number*, preserving units or labels.

the ceiling of a number

Produces the ceiling of *number*, preserving units or labels.

the square of a number

Produces number^2 .

the square root of a number

Produces the square root of *number* as a simple number.

*the [arc]sine of a number [in radians]**the [arc]cosine of a number [in radians]**the [arc]tangent of a number [in radians]*

Produces simple numbers representing various trigonometric values of *number* in degrees (by default) or radians.

the [anti]log of a number

Produces logarithmic (natural log) value of *number* as a simple number.

a number from a lower bound to an upper bound [by a step]

Produces successive numbers in the specified range by *step* (defaults to 1). All numbers must have the same units or labels.

an integer from a lower bound to an upper bound [by a step]

Produces successive integers in the specified range by *step* (defaults to 1). Each *integer* must be a simple number with an integer value.

a random number from a lower bound to an upper bound

Produces a random number within the designated bounds.

9.4 TUPLES

$\langle \text{tuple} \rangle ::= \langle [\langle \text{term} \rangle [, \langle \text{term} \rangle]^*] \rangle$

A tuple element represents an ordered sequence of elements (i.e., a vector of elements). A tuple is delimited by a pair of left and right angle brackets ($\langle \rangle$) and is composed of zero or more terms separated by commas. For example, the following are all valid tuples

```
< >
<1, 2, 3>
<the mayor, 33.5, < >>
```

When evaluated, the terms of the tuple will be evaluated in order from left to right; the resulting values will appear in the tuple. The evaluation name of a tuple consists of a left angle bracket, followed by the evaluation names of each element in the tuple separated by commas and terminated with a right angle bracket.

9.4.1 Operations on Tuples

In the following operations, *a tuple* refers to a tuple element, and *a position* refers to a simple number with a positive integer value no greater than the length of the tuple with which it appears.

a tuple is empty

Concludes true if *tuple* contains no elements, e.g.,

```
<4> If <> is empty, display yes.
YES
```

the length of a tuple

Produces the number of elements in *tuple*.

a member of a tuple [from a position]

Produces successive members of *tuple* starting with the element at *position* (defaults to 1), e.g.,

```
<5> Display the member of <a, b, c>.
A
<6> Display every member of <a, b, c>.
A
B
C
<7> Display every member of <a, b, c> from 2.
B
C
```

the member of a tuple at a position

Produces the element of *tuple* at position, e.g.,

<8> Display the member of <a, b, c> at 2.

B

<9> Display every member of <a, b, c> at 2.

B

the first member of a tuple

Produces the element in the first position of *tuple*, e.g.,

<10> Display the first member of <a, b, c>.

A

the second member of a tuple

Produces the element in the second position of *tuple*, e.g.,

<11> Display the second member of <a, b, c>.

B

the last member of a tuple

Produces the element in the last position of *tuple*, e.g.,

<12> Display the last member of <a, b, c>.

C

a tail of a tuple [from a position]

Produces successively shorter tails of *tuple* from *position* (defaults to 2), e.g.,

<13> Display every tail of <a, b, c>.

<B, C>

<C>

the tail of a tuple at a position

Produces a tuple of all elements in *tuple* from *position*, inclusive, e.g.,

<14> Display the tail of <a, b, c, d> at 3.
<C, D>

the reverse of a tuple

Produces a tuple containing the elements of *tuple* in reverse order, e.g.,

<15> Display the reverse of <a, b, c>.
<C, B, A>

the concatenation of a tuple with a tuple

Produces a tuple containing the elements of the first *tuple* followed by the elements of the second, e.g.,

<16> Display the concatenation of <a, b> with <1, 2>.
<A, B, 1, 2>

the tuple containing each <description>

Produces a tuple containing every instance of <description>, e.g.,

<17> Display the tuple containing each integer from 1 to 5.
<1, 2, 3, 4, 5>

sort a tuple in ascending order
sort a tuple in descending order

Sorts the elements of *tuple*; destructively changes *tuple*, e.g.,

<18> Let the tuple be <1, 2, 3, 4, 5>.
<19> Sort the tuple in descending order.
<20> ?
[GLOBAL Database]
 <5, 4, 3, 2, 1> IS A TUPLE.

Tuple must be a tuple of comparable numbers.

sort a tuple in ascending pair order
sort a tuple in descending pair order

Tuple must be a tuple of tuples where the first element of each component tuple is a comparable number, e.g.,

<21> Let the tuple be <<1, A>, <2, B>, <3, C>>.

<22> Sort the tuple in descending pair order.

<23> ?

[GLOBAL Database]

<<3, C>, <2, B>, <1, A>> IS A TUPLE.

9.5 STRINGS

```
<string element> ::= <string>
```

```
<string> ::= ""  
           ::= "ccc"
```

A string element is a sequence of characters delimited by double quotes ("), e.g.,

```
"This is a string"  
""
```

```
"Please respond now: "
```

Strings distinguish between uppercase and lowercase characters. Several features are being introduced in ROSIE 3.0 that greatly increase the functionality, expressiveness, and performance of strings as well as operations on strings.

In previous versions of ROSIE, the string element was simply implemented as the string data type (i.e., a one-dimensional character array or vector) of the underlying LISP system. ROSIE 3.0 implements strings as two-dimensional ragged arrays, permitting strings to be conceptualized and manipulated as rectangular blocks of text. Additional features include: fixed and free formatted strings, extended string formatting capabilities via the pattern element; augmented string syntax; automatic coercion of elements into strings; and operations for coercing strings back into other element types.

9.5.1 Formatted Strings

Strings come with a new *format* attribute. A string may be tagged as having either a *fixed*, *free*, or *mixed* format. The format of a string defines the manner in which the string will appear on an output device. There is only one significant difference between fixed and free format strings, namely, the placement of line breaks. Free format strings may contain no explicit end-of-line character; line breaks are introduced as needed to fit the text to the line length of an output device. Fixed format strings are assumed to be "user formatted"; users may specify where line breaks are to appear. Mixed format strings are composites of alternating fixed and free format strings.

Fixed format strings subsume the concept of string found in earlier releases of ROSIE. Unless otherwise specified, strings appearing in code are, by default, fixed format. Such strings are useful for developing tables or other textual structures in which the proper placement of line breaks is significant.

Free format strings contain no explicit line breaks and conform to the line length constraints of the target output device. For instance, supposing the file "testfile" had a line length of 30, then statement <3>

below would write the string to "testfile", inserting a line break before **format** to avoid running off the end of the line.

```
<2> Open "testfile" for output.  
<3> Send {Free format "Strings may be tagged  
      with a format."} to "testfile".  
<4> Close "testfile".  
<5> Type "testfile".  
Strings may be tagged with a  
format.
```

Every output device is assumed to have an associated line length accessible to ROSIE. ROSIE is not extremely sophisticated about introducing line breaks and has only limited knowledge of punctuation and no knowledge of hyphenation.

Mixed format strings enable the construction of blocks of alternating fixed and free (or free and fixed) format strings. When sent to an output device, the free format components will be contoured to fit the device, while the fixed format components will be output as is. ROSIE does *not* place a line break between alternating fixed and free format strings. If needed, such breaks must be specified by the user.

ROSIE supports the ability to coerce fixed format string into free and free format strings into fixed. However, once a fixed format string has been transformed into a free format string, there is no way to recover the formatting information (i.e., placement of line breaks, indentation, etc.) of the original string.

9.5.2 Strings and Patterns

String and pattern elements are closely related. Pattern elements (discussed in Section 9.6) describe languages of strings. When this language describes one and only one string, then the pattern can be coerced into that string.³

By the above definition, the reader may consider strings and patterns to be, at a conceptual level, instances of the same class of data element. A string is simply a pattern that describes a language consisting of itself. This is a significant notion because it permits strings to inherit the considerable expressive and representational power of patterns.

³In most instances, coercion is automatic. This is because string representation is more space efficient. Strings are easily coercible back into patterns.

9.5.3 Extended String Syntax

Among the changes to strings, ROSIE 3.0 supports an extension to the lexical syntax of strings. This extension allows strings and patterns to be specified in a succinct form that enhances both their clarity and readability.

The extended string syntax is recognized during tokenization (i.e., the lexical analysis phase of parsing ROSIE source code). When the tokenizer encounters a double quote ("), it begins reading a string. If a matching double quote is encountered before the end-of-line character, the tokenizer creates a string token of the form "ccc", where ccc are the characters appearing between the opening and closing quotes. Thus, when reading

"This is a string"

the tokenizer will recognize it as a string token.

If, however, the end-of-line character is encountered first, the tokenizer assumes an instance of the new extended string syntax, the result of which will be a sequence of tokens designating a pattern element. For example, scanning

**"East is east and west is west,
and never the twain shall meet"**

generates the tokens

```
{ "East is east and west is west" , CR ,  
  "and never the twain shall meet" }
```

These tokens would then be parsed as a pattern that is coercible into a string of the original form. Additionally, terms and subpatterns may be embedded in strings by delimiting them with matching left and right curly braces.

For a precise definition of the lexical analysis of strings, see Section 2.2.4.

9.5.4 Operations on Strings

In the following operations, *a string* refers to a string element, *a pattern* refers to a pattern element, *a file* refers to a string element that names a text file to which a channel has been open (see Chapter 11), *a name* refers to a name element, and *an element* refers to an element of arbitrary type.

Unless otherwise specified, any element can be used as *a string*; the element will automatically be coerced into a string by applying the following rules:

- 1) If the element is a pattern that describes a language of one and only one string, the element is coerced into that string.
- 2) If the element is a pattern that describes a language of more than one string, an error occurs.
- 3) If the element is any other type of element, its evaluation name is coerced into a string.

the length of a string

Produces the number of characters in *string*.

the uppercase of a string

Produces a copy of *string* with all alphabetic characters in upper case, e.g.,

```
<2> Display the uppercase of "3.4 is a real number".  
"3.4 IS A REAL NUMBER"
```

the lowercase of a string

Produces a copy of *string* with all alphabetic characters in lower case.

the string from an element

Coerces *element* into a string using the rules specified above, e.g.,

```
<3> Display the string from 'John is a man'.  
"JOHN IS A MAN"  
<4> Display the string from "{the man} is a man".  
"JOHN is a man"
```

the element from a string

Produces an element obtained by parsing *string* as *<term>* and evaluating the results, e.g.,

```
<5> Display the element from "<{the unit}, {the force}>".  
<BATTALION #5, 1600>
```

String must be syntactically recognizable as *<term>*, and its evaluation must result in an element.

the name from a string
the number from a string
the pattern from a string
the tuple from a string
the filesegment from a string
the class from a string
the description from a string
the proposition from a string
the procedure from a string

Like *the element from a string*, the above generators cause *string* to be parsed as *<term>* and evaluated, e.g.,

```

<6> Let the speed limit be 55 miles/hour
<7> Display the number from "the speed limit".
55 MILES/HOUR

```

If *string* cannot be parsed as a term or does not evaluate to the correct type of element, these generators return nothing, e.g.,

```

<8> If there is a proposition from "bogus string",
    display yes, otherwise display no.
NO

```

providing a mechanism for type checking.

evaluate a string [against timer]

Parses and evaluates *string* as *<rule>*. This means that *string* must be syntactically recognizable as such, e.g.,

```

<9> Evaluate "display hi."
HI

```

If the **against timer** option is given, then the time it takes to evaluate *string* (less parse time) is displayed after evaluation, e.g.,

```

<10> Evaluate "display hi." against timer.
HI

```

Elapsed time = 0.017 sec

send a string [to a file]

Outputs *string* to *file*. *File* must be open for output or an error occurs.

print a string [on a file]

Equivalent to **send** with the system switch **\$PRETTYFORMAT** turned on.

If *string* is specified using a pattern element, its arguments are coerced into strings without surrounding double or single quotes and output in lower case, e.g.,

```
<11> Send {'plaintiff did suffer "a loss of one eye"', cr}.
      'PLAINTIFF DID SUFFER "a loss of one eye"'
<12> Print {'plaintiff did suffer "a loss of one eye"', cr}.
      Plaintiff did suffer a loss of one eye
```

The first letter of *string* will be capitalized automatically.

print a name as a string

When **\$PRETTYFORMAT** is on, every instance of *name* will be output using *string*, e.g.,

```
<13> Print John Brown as "John Brown".
<14> Send {'the plaintiff did suffer "a loss of one eye"', cr}.
      'JOHN BROWN DID SUFFER "a loss of one eye"'
<15> Print {'the plaintiff did suffer "a loss of one eye"', cr}.
      John Brown did suffer a loss of one eye
```

match a string against a pattern

Invokes the pattern matcher to compare *string* against *pattern*. If the match succeeds, any variable bindings indicated in *pattern* are performed, otherwise, this action does nothing.

a string is matched by a pattern

Concludes true if *string* can be successfully matched against *pattern*, false otherwise. If the match succeeds, any variable bindings indicated in *pattern* are performed.

9.6 PATTERNS

```

<pattern element> ::= { <pat disj> }

<pat disj> ::= <pat conj> | <pat disj>
               ::= <pat conj>

<pat conj> ::= <subpat> , <pat conj>
               ::= <subpat>

```

A pattern element supports tasks of creating, comparing, and otherwise manipulating strings. Patterns allow programs to specify languages of strings recognizable by a finite automata. In this way, patterns represent a virtual set of strings (i.e., the set of all strings belonging to the described language). A pattern that describes a language of one and only one string can be coerced into that string. More significantly, any pattern⁴ can be coerced into an augmented form of nondeterministic finite automata (NFA) for recognizing strings that belong to its language.

Patterns were formerly a special construct in ROSIE available only as an argument to input/output (I/O) and string matching operations. In an effort to simplify and clarify the language, patterns have been reimplemented as a data primitive. Patterns remain the key construct underlying complex I/O and string manipulation operations.

A pattern is essentially an extended form of regular expression (RE). Patterns are specified as a sequence of *subpatterns* delimited by left and right curly braces ({ }) and separated by commas (,), denoting concatenation or conjunction, and vertical bars (|), denoting disjunction. One feature of patterns not found in REs is the inclusion of a form of logical variable called *pattern variables*. Pattern variables can be used to extract fields of text from strings being matched and to further constrain the language described by the pattern.

9.6.1 Generating Text

When a pattern is used to generate a string, each of its component subpatterns is coerced into a string. These substrings are concatenated in sequence to form the resulting string. For example, the patterns

```
{"The value is ", 3 + 7}
```

```
{John, " does like ", the class}
```

⁴This applies as well to any string; a string is conceptually a pattern describing a language containing only itself.

```
{ "Airfield: ", the airfield, " Target: ", the target at
that airfield, CR, " Capabilities are ", the capabilities
of that target, CR, " Vulnerability is ", the vulnerability
of that target }
```

respectively generate strings of the form,

```
"The value is 10"
```

```
"JOHN does like ACCOUNTING"
```

```
"Airfield: MIROW Target: MUNITIONS ASSEMBLY AREA
Capabilities are 100 PERCENT
Vulnerability is EXCELLENT"
```

9.6.2 Matching Text

When a pattern is used for testing whether a string belongs to the language described by the pattern, each subpattern represents a restriction on a distinct substring of the string being matched. The following examples illustrate some simple patterns and the strings they can match:

Pattern	Matches
{ "Dear ", anything (bind X), ", " }	"Dear John," "Dear Sir," "Dear Lucy Brown Butler,"
{ 3 numbers, "-", 2 numbers, "-", 4 numbers }	"563-08-4582"
{ 3 or more letters, {";" " ":" ", "}, 1 blank, 1 or more numbers, CR }	"fILE: 1245<cr>" "Los Angeles, 90025<cr>"

A pattern can be matched against a string, the characters of a text file, or input from the user's terminal.

Subpatterns are separated either by commas (,), which represent conjunction, or vertical bars (|), which represent disjunction. Logical blocks of subpatterns may be delimited by a set of curly braces. Both commas and vertical bars are right associative. Commas have a higher precedence and therefore bind more tightly than vertical bars, e.g.,

```
{a,b,c|1,2,3|x,y,z}
```

and

```
{{a,b,c}|{1,2,3}|{x,y,z}}
```

are equivalent.

A pattern variable, when specified, will be bound to the substring matching the subpattern with which it is associated. For example, when

```
("Dear ", anything (bind X), ",")
```

is matched against

```
"Dear Lucy Brown Butler,"
```

the pattern variable **X**, which is associated with the subpattern **anything**, will be bound to the string **"Lucy Brown Butler"**. Pattern variables allow programs to extract fields of text from the matched string.

Pattern variables also provide a way of further constraining the language described by a pattern. When the same pattern variable appears more than once in a pattern, the pattern will match only the target string if the pattern variable can be bound consistently (i.e., to the same substring). For example, the pattern

```
(anything (bind X), " equals ", anything (bound to X))
```

will match **"3 equals 3"** but not **"2 equals 3"**. Thus, pattern variables extend the formal descriptive power of patterns beyond regular expressions.

9.6.3 Subpatterns

```
<subpat> ::= <bind spec>

          ::= { <subpat> [, <subpat>]* }

          ::= { <subpat> [| <subpat>]* }

          ::= FREE FORMAT <subpat> [, <subpat>]*

          ::= FIXED FORMAT <subpat> [, <subpat>]*

          ::= BOX <subpat> TO WIDTH <term>

          ::= PAD <subpat>

          ::= LEFT JUSTIFY <subpat> [<dimen>]
          ::= RIGHT JUSTIFY <subpat> [<dimen>]
          ::= CENTER JUSTIFY <subpat> [<dimen>]

          ::= LJ [<term> [BY <term>]] : <subpat>
          ::= RJ [<term> [BY <term>]] : <subpat>
          ::= CJ [<term> [BY <term>]] : <subpat>

          ::= OVERLAY <subpat> ON <subpat> [<coords>] [<padding>]
```

```

::= ADJOIN <subpat> [, <subpat>]*

::= <integer> [OF] <subpat>

::= <integer> OR MOPE [OF] <subpat>
::= <integer> OR LESS [OF] <subpat>
::= <integer> OR FEWER [OF] <subpat>

::= <char class> [[NOT] IN <term>]

::= ANYTHING
::= SOMETHING

::= LINE[S]

::= RETURN[S]
::= CR[S]

::= CODES ( <integer> [, <integer>]* )

::= CHARCODE <term>

::= CONTROL <term>

::= BACKSPACE[S]
::= BS
::= BLANK[S]
::= DELETE[S]
::= END
::= EOL[S]
::= ESCAPE[S]
::= PAGE[S]
::= QUOTE[S]
::= TAB[S]

```

```

<bind spec> ::= <subpat> ( BIND TO <bind form> [AS <bind type>] )
             ::= <subpat> ( BIND <bind form> [TO <bind type>] )
             ::= BIND <subpat> TO <bind form> [AS <bind type>]
             ::= <subpat> ( BOUND TO <bind form> )

```

```

<bind type> ::= A NAME
             ::= A NUMBER
             ::= A STRING
             ::= A TUPLE
             ::= A PATTERN
             ::= A CLASS
             ::= A DESCRIPTION
             ::= A PROPOSITION
             ::= A PROCEDURE

```



```

::= A FILESEGMENT
::= AN ELEMENT

```

```

<dimen> ::= TO LENGTH <term> [AND WIDTH <term>]
        ::= TO WIDTH <term>

```

```

<coords> ::= AT < <integer> , <integer> >

```

```

<just>  ::= STARTING LEFT
        ::= STARTING RIGHT
        ::= CENTERING

```

```

<char class> ::= [NON]ALPHANUMERIC[S]
               ::= [NON]BLANK[S]
               ::= [NON]CONTROL[S]
               ::= [NON]DIGIT[S]
               ::= [NON]LETTER[S]
               ::= [NON]NUMBER[S]
               ::= [NON]NUMERAL[S]

               ::= CHARACTER[S]

```

This section describes each of the legal subpattern forms, many of which take subpatterns as arguments. Some subpatterns permit the description of languages containing more than one string; an error occurs if an attempt is made to coerce such subpatterns into a string. Other subpatterns are supplied primarily to format text; such subpatterns must (and automatically will) be coerced into a string before they can be used for matching.

Commas and Vertical Bars

The extent of a subpattern or a group of subpatterns can be delimited with a pair of curly braces ({ }). Subpatterns within curly braces can be separated by either commas (,) or vertical bars (|).

Commas can appear in subpatterns used for generating text, in which they denote concatenation, or matching text, in which they denote conjunction. Vertical bars, denoting disjunction, can appear only in subpatterns used for matching text. Commas have a higher precedence than vertical bars; both are right associative. The use of commas and vertical bars is described as follows:

```
{ <subpat> [, <subpat>]* }
```

When used in generating text, concatenates each <subpat>. If the

<subpat> are not all the same format, returns a mixed format string, otherwise returns a string of that format. Treating strings as rectangular blocks of text, concatenation appends the characters of the last line of one string to the first line of the next.

When used in matching, specifies that each <subpat> must appear in succession in the string being matched.

```
{ <subpat> [| <subpat>]* }
```

May only be used in matching operations. Specifies that one of the <subpat> must appear in the string being matched. Matching is done *independent* of <subpat> order.

Text Formatting Subpatterns

The following subpatterns are provided primarily for manipulating free and fixed formatted strings. They are intended for text generation rather than matching. They may, however, be used in pattern matching. When one of these subpatterns appears as a component of a pattern meant for matching text, it will be coerced into a string before it is passed on to the matcher. This means that such a subpattern *must* be coercible into a string, i.e., embedded subpatterns that can be used only to match text will cause an error at runtime.

Many of the following support operations on fixed format strings. For these operations, it is best to think of a fixed format string as a rectangular block of text.⁵ The number of lines in such a string is known as its *length*, and the number of characters in its longest line, its *width*. Likewise, a free format string can also be thought of as a rectangular block of text, but with length always equal to 1.

free format <subpat> [, <subpat>]*

Each <subpat> is concatenated into a single free format string. If any <subpat> is a fixed format string, it will be coerced into a free format string, i.e., user-defined line breaks will be discarded.

NOTE: When this subpattern is not the only component of a pattern, it should be delimited with a left and right curly brace, e.g.,

```
{free format subpat, . . . }
```

⁵Internally, fixed format strings are implemented as two-dimensional ragged arrays. Each row represents a line of text, and all but the last row is followed by an implicit line break--the actual end-of-line character is introduced by the output routine and does not actually appear in the string.

to avoid confusion to yourself as well as those trying to read your code.

fixed format <subpat> [, <subpat>]*

Each <subpat> is concatenated into a single fixed format string. If any <subpat> is a free format string, it will be coerced into a fixed format string--no line breaks will be inserted into this string.

NOTE: When this subpattern is not the only component of a pattern, it should be delimited with a left and right curly brace, e.g.,

{fixed format subpat, . . . }

to avoid confusion to yourself as well as those trying to read your code.

box <subpat> to width <term>

Intended for coercing free format strings into fixed format strings where no line of the string exceeds a given width.

If <subpat> is a fixed format string, generates that.

If <subpat> is a free format string, it is coerced into fixed format string. No line of the resulting string will exceed <term> characters, where <term> must evaluate to a positive integer.

If <subpat> is a mixed format string, then **box** is applied to each component, and the results are concatenated into a single fixed format string.

pad <subpat>

Intended for squaring the ragged edges of fixed format strings.

If <subpat> is a free format string, generates that.

If <subpat> is a fixed format string of width *N* generates a similar string with all lines of length *N*, padding on the right with blanks.

If <subpat> is a mixed format string, then **pad** is applied to each component string, and the results are concatenated into a new mixed format string.

(| left | right | center |) justify <subpat> [<dimen>]

<dimen> ::= to length <term> [and width <term>]
 ::= to width <term>

Intended for generating rectangular blocks of text from <subpat> in which lines are filled from the left or the right, or in which characters are centered on each line.

Generates a fixed format string with no ragged edges. The resulting string will have the dimensions specified by the **length** and **width** options, whose arguments must evaluate to positive integers. If the length or width dimensions are not given, then these values will be the length and width of <subpat>.

If <subpat> is a free or mixed format string, it will be *boxed* to fit the width of the resulting string.

Characters from each line in <subpat> are copied into the corresponding line of the resulting string. If the dimensions of <subpat> exceed the dimensions of that string, <subpat> will be truncated.

Left justification copies characters from left to right, e.g.,

{left justify "abc" to width 5}

generates "abc ". If necessary, truncates characters on the right.

Right justification copies characters from right to left, e.g.,

{right justify "abc" to width 5}

generates " abc". If necessary, truncates characters on the left.

Center justification copies characters from the center out, e.g.,

{center justify "abc" to width 5}

generates " abc ", truncating characters on the left and right as required. If a line in <subpat> will not center exactly, the odd character is pushed to the right.

LJ [<term>] [by <term>] : <subpat>
 RJ [<term>] [by <term>] : <subpat>
 CJ [<term>] [by <term>] : <subpat>

Shorthand notation for the justification subpatterns described above. The [<term>] option designates length, and [by <term>], width.

overlay <subpat> on <subpat> [<coords>] [<padding>]

<coords> ::= at <term> , <term> >

<padding> ::= **starting left**
 ::= **starting right**
 ::= **centering**

Intended for superimposing one string on top of another.

Generates a fixed format string with the dimensions of the second <subpat>. The characters of the resulting string are copied from both <subpat>; characters from the first replace characters of the second where overlap occurs, e.g.,

{overlay "Mr. John" on "Mr. Bill Brown"}

generates "Mr. John Brown".

Both <subpat> are coerced into a fixed format string if they are not already in this format.

If the <coords> option is given, it must be expressed as a tuple of two positive integers, specifying some column and row position in the final string, respectively. Once the characters of the second <subpat> are copied into the string, the characters of first <subpat> will be copied into it starting after the specified column and row position, e.g.,

{overlay "John" on "Mr. Bill Brown" at <4,0>}

generates "Mr. John Brown". The coordinates default to <0,0>, specifying the upper-left corner of the string.

The <padding> option specifies the justification for characters copied from the first <subpat>. **Starting left** copies characters from left to right, truncating on right; **starting right** copies from right to left, truncating on the left; and **centering** centers characters, truncating on either side and pushing the odd character to the right.

adjoin <subpat> [, <subpat>]*

Intended for concatenating corresponding lines from each <subpat>.

Each <subpat> that is not a fixed format string is coerced into a fixed format string.

Generates a fixed format string whose lines are the concatenation of corresponding lines of each <subpat>, e.g.,

```
{adjoin {LJ by 30:
    {free format "Hoping to trim the $130 billion
                  U.S. trade deficit, the IEEE and
                  the National Bureau of Standards
                  jointly explored the need to make"}},
{LJ by 30:
    {free format "communication with Japan, the primary
                  economic competitor of the United
                  States, a \"first priority.\" To
                  discuss these concerns, the..."}}}
```

generates

```
"Hoping to trim the $130 billion U.S. trade deficit,
the IEEE and the National Bureau of Standards jointly
explored the need to make communication with Japan, the
primary economic competitor of the United States, a
"first priority." To discuss these concerns, the..."
```

The <subpat> do not need to be the same length. Each <subpat> is treated as being the length of the longest. The empty string ("") is used as the extra lines of any <subpat> that is shorter than this.

NOTE: When this subpattern is not the only component of a pattern, it should be delimited with a left and right curly brace, e.g.,

```
{adjoin subpat, . . . }
```

to avoid confusion to yourself as well as those trying to read your code.

Text Matching Subpatterns

The next set of subpatterns allows users to describe a virtual set of strings. These subpatterns may appear only as components of patterns against which strings will be compared.

<integer> or (| **more** | **less** | **fewer** |) [of] <subpat>

Intended to match against a variable number of instances of <subpat>.

<integer> or **more** specifies <subpat> must appear at least <integer> consecutive times in the string being examined.

<integer> or **less** (or **fewer**) specifies <subpat> must appear no more than <integer> consecutive times in the string.

`<char class> [(not) in <term>]`

```

<char class> ::= [non]alphanumeric[s]
               ::= [non]control[s]
               ::= [non]digit[s]
               ::= [non]letter[s]
               ::= [non]number[s]
               ::= [non]numeral[s]
               ::= [non]blank[s]
               ::= character[s]

```

Intended for matching individual characters that fall in or out of a particular character class.

The following character classes are recognized. Each class specifies a set of characters to which a character being matched can belong. These include:

```

letter          -- a-z and A-Z;

digit           -- 0-9
number
numeral

alphanumeric    -- any letter or digit;

control         -- any control character, i.e., <ctrl>A-Z;

blank          -- the blank space character;

character       -- any character.

```

Preceding a character class with the prefix **non** designates its inverse (i.e., any character *not* in that class). The optional suffix **s** is included to enhance readability and has no other significance.

The `[(not) in <term>]` option, in which `<term>` must evaluate to a string, posts additional restrictions on the matching process. If `in <term>` is used, the characters matched must also appear as one of the characters in `<term>`, e.g.,

```
character in "0123456789"
```

is equivalent to **digit**. The inverse is true for `not in <term>`, e.g.,

```
character not in "0123456789"
```

is equivalent to **nondigit**.

anything

Equivalent to

{0 or more characters not in {EOL}}

where EOL is the end-of-line character.

something

Equivalent to

{1 or more characters not in {EOL}}

NOTE: Neither **anything** nor **something** will match beyond the end-of-line character. This is to ensure that the matcher does not scan an entire text file before discovering that a match fails. For matching beyond the end-of-line character, use the **line** subpattern.

line[s]

Equivalent to

{0 or more characters, CR}

Matches all characters up to and including a line break.

Subpatterns for Both

The next set of subpatterns specify single instances of a string and may be used for text generation and matching.

<term>

Intended to introduce arbitrary expressions into a pattern at runtime.

<term> may evaluate to any arbitrary element. Unless it returns a string or a pattern, the evaluation name of the resulting element is coerced into a string.

For generation, the value of <term> is inserted into the resulting string. If <term> evaluates to a pattern, it is first coerced into a string; if a string, it is left as is; if anything else, its evaluation name is coerced into a string.

For matching, the value of <term> is coerced into an NFA and linked into the NFA of the pattern. Thus, the value of <term> is used in matching. If this value is a pattern or string, it is turned into

an NFA, and linked into the pattern; if anything else, its evaluation name is coerced into a string, turned into an NFA and linked into the pattern.

<integer> [of] <subpat>

Indicates <integer> iterations of <subpat>.

For generation, returns the string that results from concatenating <subpat> with itself <integer> times.

For matching, <subpat> must appear in the text <integer> consecutive times.

return[s]

CR[s]

Intended for matching or generating a line break.

For generation, forces a break between lines of text. Since strings are implemented as two-dimensional arrays, this does not actually place an end-of-line character in the resulting text. Rather, it specifies that succeeding characters should appear in the next row of the array.

For matching, matches against an end-of-line character appearing in the text.

To generate a string that actually contains the end-of-line character, see **EOL** below.

quote[s]

Matches the double quote character ("), or generates a string that contains a double quote.

codes (<integer> [, <integer>]*)

Each <integer> is assumed to be an integer value representation of some character, e.g., ASCII. Matches or generates the string of characters specified by these codes.

Inverse of **charcode**.

NOTE: For the following, assume ASCII character representation.

backspace[s]
bs

Equivalent to {codes (8)}, i.e., <ctrl>H.

blank[s]

Equivalent to {codes (32)}.

end

Equivalent to {codes (4)}, i.e., <ctrl>D.

When matched against text in a file, matches the end-of-file character.

When matched against a string, matches the end of the string.

When matched against text from the terminal, matches a <ctrl>D typed by the user.

EOL[s]

Equivalent to {codes (10)}, i.e., <ctrl>J.

This subpattern should *not* be used to insert line breaks in strings. It is provided for cases, such as the **anything** and **something** subpatterns, where a string must explicitly contain the end-of-line character.

escape[s]
esc

Equivalent to {codes (27)}.

formfeed[s]
page[s]

Equivalent to {codes (12)}, i.e., <ctrl>L.

tab[s]

Equivalent to {codes (9)}, i.e., <ctrl>I.

charcode <term>

Intended for matching or generation a string of characters in their integer (machine) representation, e.g., ASCII.

Characters in <term>, which will be coerced into a string, will be converted into their integer representation. For instance, the pattern,

```
{charcode "A B C"}
```

will generate the string "6532663267", given an ASCII representation.

Inverse of **codes**.

control <term>

Intended for matching or generating a string of control characters.

Characters in <term>, which will be coerced into a string, will be converted into control characters. For instance, outputting

```
{control "G"}
```

will beep the user's terminal.

9.6.4 Pattern Variable Binding

```
<bind spec> ::= <subpat> ( BIND TO <bind form> [AS <bind type>] )
              ::= <subpat> ( BIND <bind form> [TO <bind type>] )
              ::= BIND <subpat> TO <bind form> [AS <bind type>]
              ::= <subpat> ( BOUND TO <bind form> )
```

```
<bind form> ::= <atom>
              ::= THE <description>
              ::= <description> ' S <term>
              ::= THAT <class noun>
```

```
<bind type> ::= A NAME
              ::= A NUMBER
              ::= A STRING
              ::= A TUPLE
              ::= A PATTERN
              ::= A CLASS
              ::= A DESCRIPTION
              ::= A PROPOSITION
              ::= A PROCEDURE
              ::= A FILESEGMENT
              ::= AN ELEMENT
```

Any subpattern may appear in a *bind spec*. This causes the portion of the text matched by the subpattern to be *bound* to a *pattern variable* specified by the bind spec. If the same pattern variable appears in another bind spec later in the pattern, then the matched substrings *must* be equal in order for the pattern match to succeed. If a <bind type> specification is given, then ROSIE will attempt to coerce the substring into an element of that type.

The numerous variations for the bind spec are provided to enhance readability; they are syntactic sugar and semantically equivalent. The (bound to <bind form>) bind spec is intended to be used in the second (and third, and fourth, etc.) occurrence of a pattern variable, e.g.,

{3 or more digits (bind N to a number), "-", anything (bound to N)}

will match "59483-59483", binding N to the number 59483.

9.6.4.1 Pattern Variable Specification

The <bind form> component of a bind spec designates the pattern variable. This designation may be done with either a one-word name, a description introduced by *the*, or an anaphoric reference to such a description, e.g.,

(bind N)
(bind the reply)
(bound to that reply)

When a one-word name is used, it explicitly names the pattern variable, much like the <desc var> component of description syntax names a description variable. After a successful match, the value to which this variable is bound can be referenced outside of the pattern using a rule variable of the same name.

When a description is used, it implicitly names the pattern variable via its associated description variable. This variable may be referenced anaphorically in a later bind spec. After a successful match, the value bound to the pattern variable will be stored in the database as though the action,

let the *description* be *value*

were executed, e.g.,

```
<2> Match "Mr. John Brown" against
      {"Mr. ", anything (bind the man), end} and
      display that man.
"John Brown"
<3> ?
[ GLOBAL Database ]
  "John Brown" IS A MAN.
```

The value bound to the variable can be referenced outside of the pattern as a reference to the description.

NOTE: An anaphoric term can be used only in a bind spec if it references a description used in an earlier bind spec. The reference is treated as designating the pattern variable of the earlier bind spec.

9.6.4.2 Conversion of Bound Substrings

If given, the <bind type> component tells the pattern matcher to coerce the bound substring into an element of the designated type. The default type is **string**, which requires no conversion.

Conversion to the bind type is done *after* the pattern successfully matches the text. Conversion is accomplished by parsing and evaluating the bound substring and then checking whether the resulting element matches the prescribed type.

If it does, the pattern variable will be bound to this element, and if not, the pattern match fails, e.g.,

```
<4> Match "Mr. John Brown" against
      {"Mr. ", anything (bind the man to a name), end} and
      display that man.
JOHN BROWN
<5> ?
[ GLOBAL Database ]
  JOHN BROWN IS A MAN.
```

The bind type **element** accepts any substring that can successfully be coerced into an element.

9.6.5 The Pattern Matching Process

String pattern matching is a nondeterministic, data-driven process. When a stream of characters from some input device (i.e., a string, a file, or the user's terminal) is matched against a pattern, the pattern matcher triggers success as soon as it recognizes a string belonging to the language of strings described by the pattern. If a pattern can match a string in more than one way (e.g., the pattern

```
{{3 digits, "-", anything (bind X)} |
  {anything (bind X), "-", 3 letters}}
```

could match "123-abc" in two ways, one binding X to "abc", the other binding X to "123"), then the exact manner in which the string will be recognized is *not* defined; no attempt is made to ensure "order of recognition" or preserve the ordering of disjunctive subpatterns.

ROSIE's matching operations, such as **read** and **match**, accept a pattern as an argument. They redirect I/O to the desired input stream, construct an NFA⁶ from the pattern, and pass the NFA to the pattern matcher. The pattern matcher interrogates the NFA as it reads characters from standard input, simulating a nondeterministic search through space of strings in a breadth-first manner.

The NFA is represented as a cyclic graph, the nodes of which are called *states*. There can be three types of transitions out of any state: (1) *character transitions*, which are followed upon recognizing a particular instance of a character; (2) *class transitions*, which can be followed upon recognizing an instance of a character class; and (3) *epsilon transitions*, which are not used for character recognition, but provide a flexible mechanism for linking portions of the same state that must be represented independently. A state may also be tagged as a *final state* and as either starting or ending a subpattern of some bind spec.

The pattern matcher operates by advancing a line of *travelers* through the NFA. Each traveler is advanced until one encounters a final state, which causes the match to succeed, or until the number of travelers goes to zero, which causes the match to fail. Each traveler maintains pertinent information about the particular route of the NFA it follows (e.g., its state, how pattern variables have been bound along the way, etc.).

Fetching characters one at a time until success or failure is known, the matcher enters a cycle of advancing each traveler whose state has a transition on the character. When a traveler's state has several valid transition for one character, it is cloned into enough identical travelers to follow each path. When no transition exists, the traveler is terminated. If a traveler reaches a final state, this portion of the match succeeds. If no traveler can be advanced, the match fails. The match also fails if no final state is reached after reading the end-of-file character.⁷

When a traveler traverses the states associated with the subpattern of a pattern variable, it begins recording the characters it encounters. When it exits those states, it checks to see whether it has previously recorded a binding for that variable. If not, it binds the variable to the characters collected and continues on. Otherwise, it checks the characters recently collected against the characters previously bound; if equal, it continues on, otherwise it terminates itself.

⁶The NFA cannot be further reduced to a DFA (Deterministic Finite Automata) because pattern variables require structural information that would otherwise be lost.

⁷The last character of any string is implicitly the end-of-file character.

After a traveler reaches a final state, the matcher recovers from that traveler all pattern variable bindings established during its particular traversal of the NFA. The matcher attempts to convert each bound substring into an element of the type specified in its associated bind spec. Normally, the type is **string**, which needs no conversion. If the type is other than **string**, the substring is parsed as a term, evaluated, and the type of the resulting element compared to the type of the bind spec. A mismatch in types causes the pattern match to fail.^{*} Assuming a successful conversion, the pattern variable is bound to the resulting element such that it can be referenced outside the pattern, and the pattern match ends.

Note again that through all of this, the pattern matcher never checks for ambiguities in the pattern, nor does it specify the order in which disjunctive subpatterns will be traversed. If it is important for components of a pattern to be traversed in a predefined manner, it is the responsibility of the programmer to eliminate possible ambiguities.

9.6.6 Example Application of Patterns

The two example rulesets below demonstrate how patterns can be applied to generating and processing menus of variable length.

<2> List "menu".

To generate a menu selection for a list:

Private: a menu, a count.

- [1] If the list is empty, return.
- [2] Let the count be 1 and the menu be {1}.
- [3] Send "[1] {the member of the list at 1}{cr}".
- [4] For each member of the list's tail,
 - let the count be the count + 1 and
 - the menu be {the menu | the count} and
 - send "[{the count}] {that member}{cr}".
- [5] If there is a selection from the list with the menu, produce that response.

End.

To generate the selection from a list with a menu:

Private: a reply.

Execute cyclically.

- [1] Send "{cr}Select one entry: ".
- [2] Read "{anything (bind the reply)){cr}".
- [3] Choose situation:
 - if the reply is equal to "", return;
 - if the reply is not matched by the menu,

^{*}If processing the substring results in a syntax or runtime error, the match also fails.

```

    send "{cr}Invalid response: {the reply}{cr}";
    default: produce the member of the list
              (at the number from the reply).

```

End.

<3> Display the menu selection for <Drewitz, Mirow, Parchim>.

```

[1] Drewitz
[2] Mirow
[3] Parchim

```

Select one entry: 2
Mirow

The first ruleset builds a menu from a tuple of possible choices. It passes that menu to the other ruleset that displays it and queries the user for a selection. Given a selection, the first ruleset produces the selected member of the tuple.

9.6.7 Operations on Patterns

In the following operations, *a string* refers to a string element, *a pattern* refers to a pattern element, and *a file* refers to a string element that names a text file to which a channel has been open (see Chapter 11). Additional operations on patterns that can be coerced into strings are given in Section 9.5.4.

read *a pattern* [**from** *a file*]

Reads a segment of text from *file*.

Characters are input one at a time from *file* until sufficient text has been read to

- 1) recognize an instance of *pattern*, at which time **read** returns successfully; or
- 2) recognize that no instance can be matched, at which point **read** calls an error.

Fields of the input text can only be retrieved via pattern variables.

File must be open for input or an error occurs.

match *a string* **against** *a pattern*

Invokes the pattern matcher to compare *string* against *pattern*. If the match succeeds, any variable bindings indicated in *pattern* are performed, otherwise, this action does nothing.

a string is matched by a pattern

Concludes true if *string* can be successfully matched against *pattern*, false otherwise. If the match succeeds, any variable bindings indicated in *pattern* are performed.

9.7 FILESEGMENTS

```

<filesegment> ::= ' <header> [, <rule spec>] '
               ::= ' FILE : <term> [, <header>] [, <rule spec>] '

<rule spec>   ::= <integer> [<integer>]

               ::= BEFORE <term>
               ::= AT <term>
               ::= FROM <term> TO <term>
               ::= AFTER <term>

```

A filesegment allows users to identify and manipulate rulesets, program files, and portions of program files. Filesegments are provided primarily to enable users to manipulate pieces of code through program control. Filesegments are used extensively by the file package and break package operations.

Filesegments are delimited by a pair of left quotes and optionally consist of a *file specifier*, *ruleset header*, and *rule sequence specifier*, e.g.,

```

'file: "animals"'
'file: "animals", [rule] 1'
'file: "animals", to apply a rule'
'file: "animals", to apply a rule, after [rule] 1'
'to apply a rule, from 1 to 3'

```

The first example specifies the entire contents of the program file called "animals". The second specifies only the first file rule of that file. The third specifies a ruleset from that file. The next, every rule from that ruleset between the first rule and the end statement, exclusive. The last example specifies a sequence of rules from a ruleset.

NOTE: In the last example, no file is given. In such cases, ROSIE fills in the file name automatically if the ruleset is known to the system.

9.7.1 Shorthand for Filesegments

The syntax of filesegments presented above is a formal mechanism for use in ROSIE programs. ROSIE also supports a shorthand syntax for naming filesegments. This shorthand is far more convenient and easy to use than the formal syntax when manipulating program files from the top-level monitor. Note however that only rulesets in *noticed* program files (see Chapter 13) can be specified using the shorthand notation.

With this shorthand, a file may be designated by the string that names it, e.g., the filesegment

`'file: "animals"'`

and shorthand string

`"animals"`

refer to the same program file.

A ruleset can be identified by a name element that matches some consecutive subsequence of the ruleset's name, e.g., the filesegment

`'to apply a rule'`

which is named **apply**, can be specified with any of

`apply`
`pply`
`pl`
etc.

If the shorthand matches more than one ruleset, ROSIE queries the user with each of the possibilities, e.g.,

`pl => 'file: "animals", to decide if a rule does apply (Y or N)? N`
`'file: "animals", to apply a rule (Y or N)? Y`

and allows the user to choose.

The shorthand syntax does not allow the specifications of sequences of file rules or ruleset rules. The file package operations, such as **load**, **edit**, and **list**, will accept both the formal syntax and the shorthand.

Filesegments, their application in developing and maintaining program files, and operations on filesegments are discussed further in Chapter 13.

9.8 CLASS ELEMENTS

<class element> ::= ANY <description>

A class element provides a limited deductive capability. A class element is composed of a description preceded by the function word **any**, e.g.,

any battalion
any colorless green idea
any rule which does apply to the situation

When encountered during the execution of a program, ROSIE either produces the elements named by the description or tests an element for inclusion among them.

When a class element appears as an argument of a proposition, it acts as a "wild card," matching any corresponding argument of similar propositions that belong to the implied class. In addition, the matched elements can be referenced anaphorically.

Class elements are typically used for deductive retrieval. For instance, if deciding the truth or falsity of

any man does love Mary

when the database contains

John does love Mary
John is a man

ROSIE would conclude true. After this, the matched element **John** could be referenced by **that man**. However, this is only one aspect of class elements; class elements behave quite differently for database actions that generate elements.

A class element will *never* be produced by a description. If a class element is encountered while generating the instances of a class, then, rather than producing that element, each element named by its description will be produced. This is to say, the "generate elements from a description" routine is called recursively on the class element's description, e.g.,

<4> ?
[GLOBAL Database]
MARVIN IS A MORTAL.
ANY MAN IS A MORTAL.
JOHN IS A MAN.
JOE IS A MAN.
BILL IS A MAN.

<5> Display every mortal.
MARVIN
JOHN
JOE
BILL

This is also true for elements produced from a generator ruleset.

9.8.1 Motivation and Intended Use

The class element was motivated by the need for an efficient means of retrieving specific components of affirmed propositions. For example, if we know that each woman who works in Washington D.C. works at a particular bureau, the class element **any bureau** coupled with the anaphoric term **that bureau** in

**For each woman who does work at any bureau in Washington D.C.,
display <that woman, that bureau>.**

enables us to retrieve the corresponding bureau without iterating through all bureaus in the database, e.g., as in

For each bureau,
for each woman who does work at that bureau in Washington D.C.,
display <that woman, that bureau>.

The primary application of class elements is in goal-directed backchaining to test a proposition. This function is particularly useful when much of the domain knowledge consists of a taxonomy with intended property inheritance. The class element construct makes these applications straightforward.

As an illustrative example, consider the following assertions:

**Assert any thing which is mortal will die in time.
Assert any human is mortal.
Assert any Greek is a human.
Assert Socrates is a Greek.**

While these will be stored in the physical database as

any thing which is mortal will die in time
any human is mortal
any Greek is a human
Socrates is a Greek

conceptually, a "virtual database" exists that includes the relations

Socrates is a human
any Greek is mortal
Socrates is mortal

any human will die in time
any Greek will die in time
Socrates will die in time

The reader should note that although ROSIE will conclude that such propositions in the virtual database are true, it will do so only if one of those propositions is explicitly tested for, i.e., they will not appear in the physical database as affirmed propositions. The net effect of class elements is to trade computation time for efficient allocation of memory.

9.8.2 Potential Pitfalls

There are several potential pitfalls associated with the implementation of class elements. In most applications, a user will never encounter these problems; they appear only as the use of class elements becomes more complex. This section discusses some of the ways in which class elements may create odd or erroneous behavior, and how such situations may be overcome or at least avoided.

• Recursive definitions

It is possible to define classes recursively, e.g.,

any man is a man

or, via a more circuitous route,

**any man is a human
any human is a mortal
any mortal is a man**

In either instance, a recursive definition is created that, if undetected, could result in an infinite loop and, eventually, a stack overflow error. ROSIE can detect circularities, such as those listed above, which arise from affirmed propositions, but only when there is no intermediate call to a ruleset.

When generating from a class element, ROSIE maintains a record of all classes it has generated from in uninterrupted succession. If a class element is encountered that ROSIE has already seen, it is not generated from further.

Unfortunately, it is not possible to detect all recursive definitions. If a ruleset becomes involved in the loop, e.g.,

**To generate a man:
[1] Produce any man.
End.**

ROSIE will be unable to detect the problem and enter an infinite loop.

- As arguments to rulesets

A class element, like any other type of element, can be passed as an argument to a ruleset. Once passed, it is stored in the ruleset's private database as an instance of a formal parameter. At this point, any number of problems can arise. The following are a summary of common problems:

Situation 1--

Assuming the database is empty, consider the ruleset,

To decide if a person does love a woman:

[1] If the person = any man,
 conclude true,
 otherwise conclude false.

End.

invoked by

If any man does love Mary . . .

Upon invoking this ruleset, the private database contains

Mary is a woman
Any man is a person

Executing the first and only rule of the ruleset will attempt to test whether the value of the descriptive term **the person** is equal to the element **any man**. A fatal flaw at this point is to believe that **the person** will evaluate to **any man**; rather it will try to evaluate to an instance of **man**. Since no such instance exists, it will result in an error, i.e.,

No such element exists:
THE MAN

Situation 2--

Suppose the ruleset is defined as

To decide if a person does love a woman:

[1] If the person = John,
 conclude true,
 otherwise conclude false.

End.

and the database contains

Jim is a man
 John is a man
 Jack is a man

When invoked, the **person** would evaluate to **Jim** and the test would fail.

There are two ways to have the test succeed. One way is to rewrite the first rule as

If some person = John . . .

which would generate all instances of **person** (and, thus, all instances of **man**) until it found one equal to **John**. The other is to rewrite it as

If John is a person . . .

which would subsequently test the proposition '**John is a man**' and also succeed.

Situation 3--

Assuming the situation above, when we invoked the ruleset with

If any man does love Mary . . .

what happens to the binding of **any man**'s description variable? It gets bound within the first rule of the ruleset and unbound when the ruleset terminates. This means that the element generated by **any man** cannot be passed out of the ruleset.

Situation 4--

Finally, consider

To decide if a man does love a woman:

[1] If some man = John,
 conclude true,
 otherwise conclude false.

End.

which, if invoked as before, defines the class of **man** recursively and would eventually be interpreted as defining the empty set. The equality test would, in course, fail. The only way to avoid this situation is not to use formal parameters that might conflict with classes defined outside of the private database.

While the above are only a small sample of the types of problems one might conceivably come across, this discussion should serve to demonstrate that class elements can be problematic, and their application should be undertaken with care.

9.9 INTENTIONAL DESCRIPTIONS

```

<intentional description> ::= ' THE <description> '
                           ::= ' <a/an> <description> '
                           ::= ' <description> ' S <term> '

```

An intentional description represents reference to the elements named by a description, similar in many ways to a class element. There is one key distinction between the two. As seen earlier, class elements trigger a deductive mechanism, one off-shoot of which is the inability to retrieve class elements from the the database, making class elements impossible to pass to more than one ruleset and difficult to work with in a programmatic way. Intentional descriptions are not recognized as possessing this special deductive property, and, thus, their use is not as restricted.

Intentional descriptions provide a mechanism for temporarily suspending the evaluation of descriptive terms. In a sense, intentional descriptions act as an indirect pointer to a set of data elements, conceptually serving a function that resembles "call-by-name" in ALGOL.

Intentional descriptions consist of a description, prefixed by one of the articles **a**, **an**, or **the**, and delimited by a pair of left quotes, e.g.,

```

'the equipment list'
'a command which is for time 100'
'the target's status'

```

Intentional descriptions permit system builders to represent and relate indefinite elements and generic concepts without requiring him to define explicit instances of them. The elements referenced by an intentional description may or may not exist, i.e., the set of elements described may be null.

9.9.1 INSTANCE OF...

The set of elements referenced by an intentional description can be accessed via the **instance of** construct, e.g.,

```

<2> Let <t-shirt, boots, parka, hat> be the clothing list.
<3> Display the instance of 'the clothing list'.
<T-SHIRT, BOOTS, PARKA, HAT>
<4> Assert Mirow is an airfield.
<5> Assert each of runway, munitions soft and munitions
      assembly area is a target at Mirow.
<6> Display every instance of 'a target at any airfield'.
MUNITIONS ASSEMBLY AREA
MUNITIONS SOFT
RUNWAY

```


Instance of is essentially an intentional description evaluator. It acts as a macro that expands in place to the actual description used in the intentional. The effect of this is to make the forms

instance of '*the description*'
and
description

equivalent and interchangeable. As an illustration of this consider the following.

```
<9> Assert each of Jim, Jack and John is a man.
<10> ?
[ GLOBAL Database ]
    JOHN IS A MAN.
    JACK IS A MAN.
    JIM IS A MAN.

<11> Display 'a man'.
'THE MAN'
<12> Display the instance of 'a man'.
JOHN
<13> Display every instance of 'a man'.
JOHN
JACK
JIM
<14> If John is an instance of 'a man', display yes.
YES
<15> Assert Bill is an instance of 'a man'.
<16> ?
[ GLOBAL Database ]
    BILL IS A MAN.
    JOHN IS A MAN.
    JACK IS A MAN.
    JIM IS A MAN.

<17> Deny Jack is an instance of 'a man'.
<18> ?
[ GLOBAL Database ]
    BILL IS A MAN.
    JOHN IS A MAN.
    JIM IS A MAN.

<19> Let the instance of 'the man' be George.
<20> ?
[ GLOBAL Database ]
    GEORGE IS A MAN.
```

9.9.2 The "Call-by-Name" Property

The "call-by-name" property of intentional descriptions permits users to affect global relationships programmatically. As an example, consider a generic facility for adding elements to a tuple. The action,

If the weather will be turning rainy,
include the rain gear in 'the clothing list'.

uses the intentional description 'the clothing list' to implicitly reference a tuple of elements (e.g., <t-shirt, boots>); the rain gear is an explicit reference to another element (e.g., parka). Invoking the ruleset

To include an item in a list:
[1] Let the instance of the list be
the concatenation of such an instance with <the item>.
End.

will access the instance of 'the clothing list' and modify it to include the new element. After executing the above rule, 'the clothing list' will reference a tuple containing <t-shirt, boots, parka>.

9.9.3 Operations on Intentional Descriptions

In the following operations, *a description* refers to either an intentional description or a class element, and *a database* refers to a name element identifying a database that, if optional, defaults to the *active database* (see Chapter 10).

instantiate a description to an element [in a database]

Equivalent to executing

let the instance of the description be the element

when *database* is active.

an instance of a description [in a database]

Produces successive instances of *description* from *database*.

NOTE: If the *database* option is not given, **instance of** can, if need be, call a generator ruleset. However, if *database* is given, elements will strictly be generated from *database*.

an element was [not] an instance of a description [in a database]
an element is [not] an instance of a description [in a database]
an element will [not] be an instance of a description [in a database]

These propositional forms can be used alternately to assert, deny, or test that *element* was, is, or will be an instance of *description* in *database*, e.g.,

```
<2> Assert John is an instance of 'a man'.
<3> ?
[ GLOBAL Database ]
    JOHN IS A MAN.

<4> Assert Mary is not an instance of 'a man' in beliefs.
<5> Beliefs?
[ BELIEFS Database ]
    MARY IS NOT A MAN.

<6> If John is an instance of 'a man', display yes.
YES
<7> Deny Mary is not an instance of 'a man' in beliefs.
<8> Beliefs?
[ BELIEFS Database ]
```

NOTE: If the *database* option is not used, testing these forms is equivalent to testing the propositions

```
element was [not] a description
element is [not] a description
element will [not] be a description
```

in that a predicate ruleset could be invoked to decide truth or falsity. However, if *database* is given, then a test will strictly be applied to the propositions affirmed in *database*.

increment a description [by a number] [in a database]
decrement a description [by a number] [in a database]

Alternately increments or decrements the instance of *description* in *database* as with

```
let the instance of the description in the database
    be such an instance + the number
```

9.10 INTENTIONAL PROPOSITIONS

<intentional propositions> ::= ' <proposition> '

An intentional proposition provides a means for treating propositions as data. In this way, intentional propositions capture the intent of a primitive sentence, which can then be manipulated through program control.

An intentional proposition is designated by delimiting a proposition with a pair of matching left quotes, e.g.,

'John Smith was late for work'
'the teacher did punish the student in class'
'7 is a prime number'

Since ROSIE permits programs to access, manipulate, and relate elements, intentional propositions allow users to operate with basic relations.

As an example application of propositions, consider an assertion such as

Midland Bank does believe 'Joe is interested in t-bills'

about the belief system of a bank. Two actions that use this can be stated as

If any bank does believe any thing,
consider that thing as reliable.

If any bank does believe 'any person is interested in any security'
contact that person about investing in that security.

9.10.1 IS PROVABLY...

The **is provably** construct is a proposition form for asserting, denying, and testing intentional propositions. This construct takes two arguments, an intentional proposition and a truth value (one of **true** or **false**) and alternately asserts, denies, or tests the proposition or its complement.

As a demonstration of the **is provably** construct, consider the following example pairs in which the semantics of the action in regular font is equivalent to the action in boldface preceding it:

Assert 'John is a man' is provably true.

Assert John is a man.

Assert 'John is a man' is not provably true.

Deny John is a man.

Deny 'John is a man' is not provably true.

Assert John is a man.

Assert 'John is a man' is provably false.

Assert John is not a man.

Assert 'John is a man' is not provably false.

Deny John is not a man.

If 'John is a man' is provably true, display yes.

If John is a man, display yes.

If 'John is a man' is not provably true, display yes.

Unless John is a man, display yes.

If 'John is a man' is provably false, display yes.

If John is not a man, display yes.

If 'John is a man' is not provably false, display yes.

Unless John is not a man, display yes.

Since the **is provably** forms are propositions, they can also be used as constituent relations of an intentional proposition, e.g.,

Assert 'John is a man' is not provably true' is provably false.

which asserts 'John is a man'.

9.10.2 Operations on Intentional Propositions

In the following operations, *a proposition* refers to an intentional proposition, and *a database* refers to a name element identifying a database that, when used as an optional argument, defaults to the *active database* (see Chapter 10).

an affirmed proposition [from *a database*]

Successively produces every affirmed proposition from *database* as an *intentional proposition* element.

assert *a proposition* [in *a database*]

add *a proposition* to *a database*

Asserts *proposition* in *database*.

deny *a proposition* [from *a database*]
remove *a proposition* from *a database*

Denies *proposition* from *database*.

a proposition is [not] provably true
a proposition is [not] provably false

When used as a predicate,

proposition is provably true

concludes true if *proposition* can be proved true from assertions in the database or from a predicate ruleset, false otherwise; and

proposition is provably false

concludes true if the complement of *proposition* can be proved true from assertions or a predicate ruleset, false otherwise, e.g.,

If 'John is a man' is not provably true, . . .

Unless John is a man, . . .

If 'John is a man' is provably false, . . .

If John is not a man, . . .

When asserted,

proposition is provably true

asserts *proposition*, and

proposition is not provably true

denies *proposition*, and

proposition is [not] provably false

likewise asserts or denies the complement of *proposition*, e.g.,

Assert 'John is a man' is provably true

Assert John is a man

Assert 'John is a man' is not provably true

Deny John is a man

Assert 'John is a man' is provably false

Assert John is not a man

Assert 'John is a man' is not provably false

Deny John is not a man

When denied,

proposition is provably true

denies *proposition*;

proposition is not provably true

asserts *proposition*; and

proposition is [not] provably false

likewise denies or asserts the complement of *proposition*, e.g.,

Deny 'John is a man' is provably true

Deny John is a man

Deny 'John is a man' is not provably true

Assert John is a man

Deny 'John is a man' is provably false

Deny John is not a man

Deny 'John is a man' is not provably false

Assert John is not a man

a proposition is [not] true [in a database]

a proposition is [not] false [in a database]

Like **is provably** with the addition that *database* is activated before *proposition* or its complement is asserted, denied, or tested.

NOTE: Unlike the **is provably** predicate, these forms will not invoke a predicate ruleset to prove or disprove *proposition*; such proof must come strictly from the assertions in *database*.

a proposition is negated

Concludes true if *proposition* is negated (i.e., contains the word **not**), true otherwise, e.g.,

<2> If 'John does not love Mary' is negated, display yes.
YES

the query from a proposition

Produces a string in which *proposition* is restated as a question, e.g.,

<3> Display the query from 'John is a happy man'.
"IS JOHN A HAPPY MAN?"

NOTE: If *proposition* is negated, the negation is ignored in the resulting query, e.g.,

<4> Display the query from 'John is not a happy man'.
"IS JOHN A HAPPY MAN?"

9.11 Intentional Procedures

217

9.11 INTENTIONAL PROCEDURES

`<intentional procedure> ::= ' <procedure> '`

An intentional procedure enables users to treat the procedure action type as an element of data. Intentional procedures capture the intent of unexecuted actions, which can then be manipulated through program control and executed at a later time.

An intentional procedure is designated by delimiting a procedure with a pair of matching left quotes, e.g.,

```
'move USS Nimitz from Le Havre to New York'
'broadcast the report'
'rendezvous with the strike unit'
```

As an example application of intentional procedures, consider a program that queues actions by some time metric, e.g.,

```
Assert 'move USS Nimitz from Le Havre to New York' is
      an action to execute at time 100.
Assert 'broadcast the report' is an action to execute
      at time 120.
Assert 'rendezvous with the strike unit' is an action
      to execute at time 150.
```

and executes actions in the queue, e.g.,

```
For each action to execute at the current time,
  execute that action.
```

9.11.1 Operations on Intentional Procedures

In the following, *a procedure* refers to an intentional procedure element.

execute a procedure

Executes *procedure*, e.g.,

```
<5> Execute 'display hi donna'.
HI DONNA
```

X. THE DATABASE MECHANISM

The initial, intermediate, and final results of ROSIE programs are stored as *affirmed* propositions in ROSIE's database. Propositions can be asserted (affirmed in the database) and denied (removed from the database). It is possible to test the truth or falsity of a proposition against the contents of the database as well as generate the members of a class defined by affirmed class relations (i.e., propositions using the *is-a* copula).

ROSIE's database structure actually consists of two conceptually separate layers. The first is the *physical database*, which contains affirmed propositions. The second is the *virtual database*, which consists of those relations that can be computed from other relations via ruleset invocation or a limited deductive retrieval mechanism provided with class elements.

10.1 THE PHYSICAL DATABASE

The physical database is used to store propositions. Propositions are affirmed under a three-valued logic system, i.e., a proposition is either true, false, or unknown. Additionally, the physical database can consist of up to three separate databases at a time. These databases are tiered so that propositions in one may hide propositions in another. ROSIE also allows users to create alternate databases, which may be swapped in and out of context by program control.

10.1.1 Three-Valued Logic

Propositions are affirmed in the physical database using a three-valued logic system. Within such a system, if a proposition or its negation is affirmed, then the proposition is provably true or false, respectively. Otherwise, the proposition is unknown and has an indeterminate truth value.

This style of three-valued logic provides ROSIE with an "open-world" assumption. It implies that ROSIE may not have complete knowledge about a particular situation. In such cases, truth or falsity will not be inferred from the absence of contradictory information.¹

¹This does not mean, however, that assumptive behavior will not appear in a program. For instance, the action,

If John does like Mary, let John be Mary's boyfriend,
otherwise let any man be Mary's boyfriend.

executes its *then-part* only if the condition succeeds. If the condition does not succeed, then the *else-part* is executed; i.e., ROSIE does *not* test if the proposition is provably false.

10.1.2 Database Actions

ROSIE provides four primitive actions for manipulating the physical database: **assert**, **deny**, **let**, and **create**. A user adds to the database by asserting propositions, e.g.,

assert USS Nimitz is docked at Le Havre

by assigning, via **let**, a distinct value as the one and only instance of a class, e.g.,

let the objective be Red River Crossing

or by creating a generic instance with **create**, e.g.,

create a strategic command center

Propositions are removed from the database by denial, e.g.,

deny USS Nimitz is docked at Le Havre

Note that denial is *not* retroactive, i.e., if the denied proposition was not affirmed in the first place, the **deny** action will have no latent effect if the proposition is affirmed at a later time.

10.1.3 Contradictory Assertions

The physical database is automatically kept consistent in regard to simple contradictions. A simple contradiction occurs when the complement of a proposition being asserted is already affirmed in the database. In such cases, the affirmed complement is discarded in favor of the new assertion, e.g.,

<2> Assert USS Nimitz is docked at Le Havre.

<3> ?

[GLOBAL Database]

USS NIMITZ IS DOCKED AT LE HAVRE.

<4> Assert USS Nimitz is not docked at Le Havre.

<5> ?

[GLOBAL Database]

USS NIMITZ IS NOT DOCKED AT LE HAVRE.

ROSIE's ability to check for inconsistencies in the physical database is limited to immediately comparable propositions. It does not include reasoning through the effects of rulesets or virtual relations, e.g.,

<6> Assert John does like any woman.

<7> Assert each of Mary and Sara is a woman.

<8> If John does like Mary, display yes, otherwise display no.

YES

<9> Assert John does not like Mary.

<10> ?

[GLOBAL Database]

JOHN DOES NOT LIKE MARY.

JOHN DOES LIKE ANY WOMAN.

SARA IS A WOMAN.

MARY IS A WOMAN.

<11> If John does like Mary, display yes, otherwise display no.

NO

<12>

In this interaction, ROSIE will not catch the contradiction of statements <6> and <9>. In such cases, the order of assertions ultimately defines the truth value of the proposition in question.

10.1.4 Alternate Databases

Occasionally, a method is needed for partitioning data, i.e., storing different facts in different databases. This may arise because we wish to model multiple points of view or because we want to restrict attention momentarily to a subset of those facts that are most relevant. To support such needs, ROSIE allows users to create *alternate databases* and specify when they should be brought in and out of context.

10.1.4.1 Naming and Creating Databases

Every database has a name by which it can be identified. This can be any name element; multiword database names are allowed. ROSIE comes with a predefined database, named **global**, which functions as *the global database*. The private database of a ruleset invocation (see Section 4.2.3.3) is named **private**. An alternate database is named by the user when created.

The user can create and subsequently name an alternate database with the **activate** procedure, e.g.,

activate conclusions

which makes that database *the active database*. If no such database exists, one by that name is created.

10.1.4.2 The Global, Active, and Private Databases

The physical database is structured such that three databases can be in context at any given time. Conceptually, one can imagine that the physical database has three slots that can be filled to form a composite database. These slots are tiered, one on top of the other. When accessing the physical database, ROSIE moves down through the database in the topmost slot to the database in the lowest slot. This has the

effect that information at one level could obscure information at another.

The slot at the lowest level is reserved for the global database, which is always present. The slot at the highest level is reserved for the private database of a ruleset invocation and changes as rulesets come in and out of context. The middle slot is reserved for the active database. This slot can be filled with any alternate database specified by the user with the **activate** procedure. Thus, when searching the database, ROSIE firsts examines the private database, then the active, and then the global.

The global database is created whenever a ROSIE session is initiated. It is intended to contain information that remains constant throughout the execution of a program. When no alternate database is active, the global database is treated as the active database.

Each ruleset invocation is allocated a private database. Whenever an invocation is in context (i.e., has not been suspended by the invocation of another ruleset) its private database becomes *the* private database. A private database and the relations asserted into it are discarded upon termination of the invocation. As noted in Section 4.2.3.3, there are a number of restrictions on the use of the private database. In addition, the private database cannot be explicitly activated or deactivated.

The active database is maintained and controlled by the user's program. A user can bring alternate databases in and out of an active role with the **activate** and **deactivate** procedures. For instances,

activate conclusions

makes the alternate database named **conclusions** the active database. If this is followed by

activate beliefs

the current contents of the active database will be stored under the name **conclusions**, and the data stored under the name **beliefs** will become active. The call

deactivate

will deactivate **beliefs** without activating any other alternate database, making the global database active.

An alternate database can be "temporarily" activated from within a ruleset with the **swap in** procedure. **Swap in** remembers which database was active at the time it was called. When the ruleset invocation terminates, the original active database is reactivated automatically. This is true even if the invocation is terminated due to an error, making **swap in** an effective method for restoring system state.

10.1.4.3 Accessing the Physical Database

The database actions **assert** and **deny** are restricted to the active database (and, in a limited form, to the private database). Tests are made against the entire physical database. There is a set of database operations such as

add a proposition to a database
and
remove a proposition from a database

that allows programs to specify the database in which particular database actions are to be executed.

As an illustrative example, consider the following sample session:

```
(R)
[ ROSIE Version 3.0 (PSL) 26-May-86 ]

<2> Assert each of Jim, Jack and John is a man.
<3> Assert each of Mary and Sara is a woman.
<4> Assert any man does like any woman.
<5> ?
[ GLOBAL Database ]
  ANY MAN DOES LIKE ANY WOMAN.
  SARA IS A WOMAN.
  MARY IS A WOMAN.
  JOHN IS A MAN.
  JACK IS A MAN.
  JIM IS A MAN.

<6> Display every man.
JOHN
JACK
JIM
<7> Activate beliefs.
<8> ?
[ BELIEFS Database ]

<9> Global?
[ GLOBAL Database ]
  ANY MAN DOES LIKE ANY WOMAN.
  SARA IS A WOMAN.
  MARY IS A WOMAN.
  JOHN IS A MAN.
  JACK IS A MAN.
  JIM IS A MAN.

<10> Display every man.
JOHN
JACK
JIM
```

```

<11> If John does like Sara, display yes.
YES
<12> Assert John does not like Sara.
<13> ?
[ BELIEFS Database ]
    JOHN DOES NOT LIKE SARA.

<14> If John does like Sara, display yes.
<15> Deactivate.
<16> ?
[ GLOBAL Database ]
    SARA IS A WOMAN.
    MARY IS A WOMAN.
    JOHN IS A MAN.
    JACK IS A MAN.
    JIM IS A MAN.

<17> If John does like Sara, display yes.
YES
<18> Beliefs?
[ BELIEFS Database ]
    JOHN DOES NOT LIKE SARA.

```

Statements <2> through <4> initialize the global database. Statement <5> demonstrates use of the ? command to examine the contents of the active database, initially **global**. Statement <6> accesses elements of the database.

In statement <7>, we activate an alternate database named **beliefs**. When we examine the contents of the active database in <8>, we find that it is empty. Statement <9> shows, however, that our initial assertions are still present in the global database, and statements <10> and <11> show that these assertions are still accessible. Statements <12> through <14> illustrate how we can overload relations in the global database.

Statement <15> deactivates **beliefs** database, reactivating **global** database. While statements <16> and <17> demonstrate that our actions in the **beliefs** database had no ill effects on the contents of **global**, statement <18> shows that we have not lost those assertions.

10.2 THE VIRTUAL DATABASE

The virtual database provides support for relations that either cannot or should not be described by affirmed propositions. For instance, relations such as '**3 is greater than 2**' are more practical to compute than store. The virtual database consists of both predicate and generator rulesets, and *virtual relations*.

Predicate and generator rulesets, discussed in Chapter 4, allow users to define subroutines for alternately deciding the truth or falsity of a proposition, or producing the elements of a class. Virtual relations, which are affirmed propositions containing a class element argument, e.g.,

any ship is a vessel

give users a method for specifying relations that hold over a range of elements.

The primary trade-off between the physical and virtual database is time versus space. In general, relations stored explicitly in the physical database require more memory than equivalent relations from the virtual database. Alternatively, relations derived from the virtual database require extended computation for their retrieval.

10.2.1 Predicate and Generator Rulesets

One component of the virtual database are relations that can be computed by predicate and generator rulesets. These rulesets are invoked automatically when the physical database is unable to satisfy a request for information. Their results are treated as though formulated from relations in the physical database.

Predicate rulesets provide a means of deciding the truth or falsity of a proposition through direct computation. When ROSIE is unable to prove or disprove a proposition from affirmed propositions in the database, ROSIE looks for a predicate ruleset for testing the proposition or its complement. If such a ruleset exists, it is invoked automatically and its resulting conclusion, or failure to return a conclusion, decides the outcome of the test. For more information, see Section 4.2.2.2.

Generator rulesets are used to produce a stream of elements belonging to a class. When generating instances of a class, ROSIE first produces all elements that satisfy a relation of the form

element is a class

in the physical database. Once all such elements have been exhausted, ROSIE searches the virtual database for a ruleset capable of generating additional elements of the class, invoking this ruleset if it exists. For further details on generator rulesets, see Section 4.2.2.3; for further details about element generation, see Chapter 7.

Rulesets, when defined, are stored in a single structure, accessible globally and indexed by their header. When ROSIE looks up a ruleset, it examines the global ruleset store for the appropriately identified ruleset. While individual rulesets may be enabled and disabled by program control, there is no way to scope or otherwise partition the ruleset store.

10.2.2 Virtual Relations

A virtual relation is a proposition that specifies a relationship that holds for a class of elements, rather than for a single specific element, e.g.,

any man is mortal

as opposed to

Jim is mortal
Jack is mortal
John is mortal
etc.

As observable in the above example, the key feature of a virtual relation is the presence of a class element (e.g., **any man**). If a class element appears as an argument to an affirmed proposition in the physical database, or if a class element is embedded within an argument of such a proposition, then that proposition is said to describe a virtual relation. ROSIE recognizes class elements as implicit references to any element that can be generated from its base description. A proposition that contains a class element holds for any element referenced by that class.

As an illustrated example, consider the following sample session:

```
(R)
[ ROSIE      Version 3.0  (PSL)  26-May-86 ]

<2> Assert each of Jim, Jack and John is a man and
      each of Mary and Sara is a woman.
<3> Assert any man does like any woman.
<4> Assert John does not like Sara.
<5> Assert any woman is a person and any man is a person.
<6> ?
[ GLOBAL Database ]
  JOHN DOES NOT LIKE SARA.
  ANY MAN DOES LIKE ANY WOMAN.
  ANY MAN IS A PERSON.
  ANY WOMAN IS A PERSON.
  SARA IS A WOMAN.
  MARY IS A WOMAN.
  JOHN IS A MAN.
  JACK IS A MAN.
  JIM IS A MAN.

<7> If John does like Jack, display yes.
<8> If John does not like Jack, display yes.
<9> If John does like Mary, display yes.
YES
```

```

<10> If John does like Sara, display yes.
<11> If John does not like Sara, display yes.
YES
<12> Display every person.
SARA
MARY
JOHN
JACK
JIM
<13> Display every person who does like Mary.
JOHN
JACK
JIM
<14> Display every person who does like Sara.
JACK
JIM
<15> Display every person who does like any person.
JOHN
JACK
JIM
<16> Display every person who does not like any person.
JOHN

```

Statements <2> through <5> initialize the database, defining a class of **man**, **woman**, and **person**, as well as specifying a **does like** relation that holds between instances of **man** and **woman**. Statements <7> through <11> demonstrate the virtual relations in proving the truth or falsity of propositions. Finally, statements <12> through <16> demonstrate how virtual relations behave when used to generate instances of a class.

The example above points out an important aspect of class elements, namely, delayed evaluation. If we had said in statement <5>

Assert every woman is a person and every man is a person.

then the propositions,

```

SARA IS A PERSON
MARY IS A PERSON
JOHN IS A PERSON
etc.

```

would have appeared in the database when we made the query of statement <6>. Since class elements act as placeholders for the individual instances of a class, the effect of using the **any** construct is to delay enumeration of all pertinent *is-a* relations until needed. This is an important property when all elements of a class are not known a priori.

A final observation that arises from this example is the application of virtual relations as "default" relations. By placing the assertion in statement <3> before any other assertion of the **does like** relation, we have effectively made it the default; it will be the last relation encountered when testing the truth or falsity of any proposition concerning the **does like** relation, allowing us to test for exceptions such as in statement <11>.

10.3 ASSERTING, TESTING, AND DENYING PROPOSITIONS

In this section, we provide a somewhat more precise definition of what it means to assert, deny, and test a proposition. Whenever asserting, denying, or testing a proposition, ROSIE first evaluates its the arguments, i.e., its subject, object, and the objects of its prepositions. ROSIE follows the appropriate procedures described below.

NOTE: If the proposition specifies a class relation, e.g.,

Red River Crossing is an objective which is strategic

then it potentially specifies a set of propositions, i.e., the base *is-a* relation, e.g.,

Red River Crossing is an objective

and the relations in the relative clause, e.g.,

Red River Crossing is strategic

In such cases, the following procedures are applied to each proposition in that set, starting with the base *is-a* relation.

Assertions--

To assert a proposition (initiating an assert event), ROSIE does the following:

- 1) It checks for an assert demon monitoring assert events of the given proposition, invoking the demon if it exists.
- 2) If there is no such demon or if the demon's invocation is terminated by the **continue** procedure, ROSIE proceeds to the next step, otherwise the assert event is aborted.
- 3) if the proposition or its complement has already been affirmed in the active database, ROSIE discards the existing assertion.
- 4) ROSIE adds the given proposition to the active database, making it the most recent assertion.

Denials--

To deny a proposition (initiating a deny event), ROSIE does the following:

- 1) It checks for a deny demon monitoring deny events of the given proposition, invoking the demon if it exists.
- 2) If there is no such demon or if the demon's invocation is terminated by the **continue** procedure, ROSIE proceeds to the next step, otherwise the deny event is aborted.
- 3) If the given proposition has been affirmed in the active database, ROSIE removes it, otherwise, no action occurs.

Tests--

To test a proposition (initiating a test event), ROSIE does the following:

- 1) It checks for a test demon monitoring test events of the given proposition, invoking the demon if it exists.
- 2) If there is no such demon or if the demon's invocation is terminated by the **continue** procedure, ROSIE proceeds to the next step, otherwise the test fails.
- 3) ROSIE tries to prove or disprove the proposition by examining all affirmed propositions in order of recency.
 - a) If the proposition being tested is affirmed, or if it can be deduced from a virtual relation, then the test succeeds.
 - b) Alternatively, if the complement of the given proposition is affirmed, or can be inferred from a virtual relation, then the test fails.
- 4) If none of the above conditions is true, ROSIE looks up a predicate ruleset that tests for the proposition or its complement--only one or the other can be defined at one time. If such a ruleset exists, it is invoked and its results decide the truth or falsity of the proposition as follows:
 - a) If predicate tests for the proposition and concludes true, then the test succeeds. If it concludes false or returns without making a conclusion, then the test fails;
 - b) Alternatively, if the predicate tests for the proposition's complement and concludes false, then the test succeeds. Otherwise, the test fails.

NOTE: When examining all affirmed propositions in step 3 above, ROSIE is working with a *partially closed* database and will consider only those propositions that are similar to the test proposition (i.e., which use the same verb form and associated prepositions) and which were affirmed at the start of the test event. This means that adding or removing propositions from this set during the test will not affect its outcome. For example, consider the following interactions

<10> List "test".

To decide if a person is a man:

[1] Create a person who does like Mary.

[2] Conclude false.

End.

<11> ?

[GLOBAL Database]

JOHN DOES LIKE MARY.

<12> If any man does like Mary, display that man.

<13> ?

[GLOBAL Database]

PERSON #1 DOES LIKE MARY.

JOHN DOES LIKE MARY.

PERSON #1 IS A PERSON.

When line <12> compares 'any man does like Mary' to the affirmed proposition 'John does like Mary' it calls the ruleset seen in line <10> to decide if 'John is a man'. The ruleset concludes false, but it also affirms a new instance of the **does like** relation. Obviously, this has the potential for causing the test in line <12> to continue indefinitely, but because the test event is closed over the **does like** relation, the new relation is not considered and the test halts.

10.4 AUTO-QUERY MODE

ROSIE provides a facility for automatically querying an outside data source (such as the user) when the truth or falsity of a proposition cannot be determined. This mechanism is controlled by the system switch **\$AUTOQUERYFLG** (the default setting is off).

When **\$AUTOQUERYFLG** is on, and ROSIE is not able to prove or disprove the truth or falsity of a proposition, ROSIE will look for the predicate ruleset

To decide if a proposition is confirmed:

If such a ruleset exists, it will be applied to the positive form of the proposition. The conclusion of the ruleset decides the truth value of the proposition.

ROSIE provides a default query mode predicate defined as

To decide if a proposition is confirmed:

Private: a reply.

Execute cyclically.

[1] Send "{cr}{the query for the proposition} ".

[2] Read "{anything (bind to the reply)}{cr}".

[3] Select the uppercase of the reply:

<"YES"> assert the proposition is provably true and
conclude true;

<"NO"> assert the proposition is provably false and
conclude false;

<" "> return;

default: send "{cr}Please respond YES or NO{cr}".

End.

To see how this facility works, consider the following sample session.

```

(R)
[ ROSIE      Version 3.0  (PSL)  26-May-86 ]

<2> ?
[ GLOBAL Database ]

<3> If John is a man, display yes.
<4> Switch on $AUTOQUERYFLG.
<5> Redo 3.

IS JOHN A MAN? y

Please respond YES or NO

IS JOHN A MAN? yes
YES
<6> ?
[ GLOBAL Database ]
    JOHN IS A MAN.

<7> Redo 3.
YES
<8> If John does not love Mary, display yes.

DOES JOHN LOVE MARY? yes
<9> ?
[ GLOBAL Database ]
    JOHN DOES LOVE MARY.
    JOHN IS A MAN.
```

With this definition, the auto-query mechanism allows ROSIE to build up its database by consulting the user. While this mechanism is not always appropriate, it is extremely useful for diagnostic tasks.

10.5 DATABASE OPERATIONS

In the following operations: *A database* refers to a name element identifying a database; *a file* refers to a string element that identifies a text file to which a channel has been open (see Chapter 11); *a proposition* refers to an intentional proposition; *a description* refers to an intentional description; and *an element* refers to any arbitrary element. Unless otherwise specified, these operations cannot be applied to the private database of a ruleset invocation. Operations that take an optional database argument will be applied to the active database by default. If no alternate database is active, the global database is treated as the active database.

activate [*a database*]

Sets the active database to be *database*. If no such database exists, one is created by that name. If *database* is not given, the global database is activated.

deactivate

Deactivates the active database, if any, and activates the global database.

swap in *a database*

Temporarily activates *database*.

If executed within a ruleset, the active database will be reset to its original value when the ruleset invocation terminates.

If used in a monitor rule, resets the active database upon execution of the rule.

the active database

Produces the name of the active database, or GLOBAL, if no database is active.

a database

Successively produces the names of existing databases, including the

global database but not including the private database. The name of the active database will be produced first.

an alternate database

Successively produces the names of existing alternate databases, not including the global database. The name of the active database will be generated first.

show [a database]

Displays every affirmed proposition in *database*. If the name **private** is given, displays the contents of the active private database.²

?

Equivalent to **show**; a shorthand for use within the top-level monitor or a break monitor.

<name element>?

Equivalent to **show** <name element>; a shorthand for use within the top-level monitor or a break monitor. **Private?** displays the contents of the private database of a ruleset invocation.

NOTE: This syntax supersedes the older <term>? syntax, which was equivalent to **describe** <term>.

clear a database
clear database

Removes all affirmed propositions from *database*. If **database** is given, clears the active database.

NOTE: This operation does not remove propositions using **deny** and, thus, will not invoke a deny demon.

dump [a database] as a file

Stores (in a machine readable format) the contents of *database* into *file .db* (i.e., a file whose name is created by appending a **.db**

²Only applies during a ruleset invocation.

extension to *file*).

restore a file [to a database]

Undoes **dump**. Reads the contents of a database from *file.db* and makes them the contents of *database*, discarding the older contents of *database* if any.

An error occurs if *file.db* does not exist.

copy to a database

Copies the contents of the active database to *database*, destroying anything that was in *database*.

copy from a database

Copies the contents of *database* into the active database, destroying anything that was in the active database.

If *database* does not name an existing database, equivalent to **clear database**.

forget about an element [in a database]

Removes all propositions from *database* that use *element* as a top-level argument--does not check if *element* is embedded in an argument.

describe an element [in a database]

Displays all propositions from *database* that use *element* as a top-level argument.

an affirmed proposition [from a database]

Successively produces every affirmed proposition from *database* as an intentional proposition.

assert <proposition> [and <proposition>]*

Affirms each <proposition> in the active database.

assert *a proposition* [*in a database*]
add *a proposition to a database*

Asserts *proposition* in *database*.

deny <proposition> [**and** <proposition>]*

Removes each <proposition> from the active database.

deny *a proposition* [**from a database**]
remove *a proposition from a database*

Denies *proposition* from *database*.

create <a/an> <description>

Creates an instance of <description>. This instance will be a name element generated by appending #*N* to the class noun of <description>, where *N*, a positive integer associated with the class noun, is incremented by one for each name so created.

This element is asserted as an instance of <description> in the active database, e.g.,

```
<3> Create a happy man.
<4> ?
[ GLOBAL Database]
  MAN #1 IS A HAPPY MAN.
```

let <let form> [**and** <let form>]

```
<let form> ::= the <description> be <term>
             ::= <term> ' s <description> be <term>
             ::= <term> be the <description>
             ::= <term> be <term> ' s <description>
```

Makes the value of <term> the singular instance of <description> in the active database, e.g.,

Let the counter be 1

is conceptually equivalent to executing

Deny every counter is a counter and assert 1 is a counter

NOTE: <term> and <description> can be arranged in any order; in the case of

Let the desc #1 be the desc #2

desc #1 is treated as <description> and the desc #2 as <term>.

instantiate a description to an element [in a database]

Equivalent to executing

let the instance of *description* be *element*

when *database* is active.

an instance of a description [in a database]

Produces successive instances of *description* from *database*.

NOTE: If the *in database* option is not given, *instance of* can, if need be, call a generator ruleset. However, if *database* is given, elements will strictly be generated from *database*.

an element was [not] an instance of a description [in a database]

an element is [not] an instance of a description [in a database]

an element will [not] be an instance of a description [in a database]

These propositional forms can be used alternately to assert, deny, or test that *element* was, is, or will be an instance of *description* in *database*, e.g.,

<2> Assert John is an instance of 'a man'.

<3> ?

[GLOBAL Database]

JOHN IS A MAN.

<4> Assert Mary is not an instance of 'a man' in beliefs.

<5> Beliefs?

[BELIEFS Database]

MARY IS NOT A MAN.

<6> If John is an instance of 'a man', display yes.

YES

<7> Deny Mary is not an instance of 'a man' in beliefs.

<8> Beliefs?

[BELIEFS Database]

NOTE: If the *in database* option is not used, testing these forms is equivalent to testing the propositions

element was [not] a description
element is [not] a description
element will [not] be a description

in that a predicate ruleset could be invoked to decide truth or falsity. However, if *database* is given, then a test will strictly be applied to the propositions affirmed in *database*.

increment a description [by a number] [in a database]
decrement a description [by a number] [in a database]

Alternately increments or decrements the instance of *description* in *database* as with

let the instance of the description in the database
be such an instance + the number

a proposition is [not] provably true
a proposition is [not] provably false

When used as a predicate,

proposition is provably true

concludes true if *proposition* can be proved true from assertions in the database or from a predicate ruleset, false otherwise; and

proposition is provably false

concludes true if the complement of *proposition* can be proved true from assertions or a predicate ruleset, false otherwise, e.g.,

If 'John is a man' is not provably true, . . .
 Unless John is a man, . . .

If 'John is a man' is provably false, . . .
 If John is not a man, . . .

When asserted,

proposition is provably true

asserts *proposition*, and

proposition is not provably true

denies *proposition*, and

proposition is [not] provably false

likewise asserts or denies the complement of *proposition*, e.g.,

Assert 'John is a man' is provably true

Assert John is a man

Assert 'John is a man' is not provably true

Deny John is a man

Assert 'John is a man' is provably false

Assert John is not a man

Assert 'John is a man' is not provably false

Deny John is not a man

When denied,

proposition is provably true

denies *proposition*, and

proposition is not provably true

asserts *proposition*, and

proposition is [not] provably false

likewise denies or asserts the complement of *proposition*, e.g.,

Deny 'John is a man' is provably true

Deny John is a man

Deny 'John is a man' is not provably true

Assert John is a man

Deny 'John is a man' is provably false

Deny John is not a man

Deny 'John is a man' is not provably false

Assert John is not a man

a proposition is [not] true [in a database]

a proposition is [not] false [in a database]

Like **is provably** with the addition that *database* is activated before *proposition* or its complement is asserted, denied, or tested.

NOTE: Unlike the **is provably** predicate, these forms will not invoke a predicate ruleset to prove or disprove *proposition*; such proof must come strictly from the assertions in *database*.

XI. INPUT/OUTPUT

The input/output (I/O) operations allow programs to read from and write to text files, communicate with the user's terminal, and initiate jobs on the host operating system. Users can also create transcript files that record all or part of a ROSIE session.

11.1 CHANNELS

All I/O passes through an internal data structure called a *channel*. A channel is a line to a file device through which input can be read or output written. The user's terminal is one such device; text files in the host directory system are another. Where the host operating system (OS) permits, a special channel is available for sending commands to the OS and reading the results.

Input from a channel is performed using the **read** procedure, which takes a pattern element as one of its arguments. Characters are read one at a time until the pattern has been matched. Pattern variables provide a means of extracting substrings of the input text.

Output to a channel is performed using either the **display**, **send**, or **print** procedures. **Display** simply outputs the evaluation name of any given element to the standard output channel, while **send** outputs a formatted string to some specified channel. **Print** is a specialized form of **send** that attempts to "beautify" the string before output.

11.1.1 Opening and Closing Channels

Before a file can be accessed for input or output, a channel must be open to it. The channel can be open for reading or writing, but not both simultaneously. Exceptions to this are *the TTY channel* and *the OS channel*, which are special channels open for both reading and writing.

A channel is created using the **open** procedure. For instance, the action,

Open "mydata" for input.

opens an input channel to the text file "mydata", and

Open "myresults" for output.

opens an output channel to the text file "myresults".¹

¹If the file did not exist before, it will be created. If it did exist, the old contents will be destroyed.

Once open, a channel assumes the name of the file. Therefore, in the above examples, we can refer to the channels open as **"mydata"** and **"myresults"**, respectively. To perform I/O through a channel explicitly, its names must be passed as an argument to the particular I/O operation being applied.

ROSIE does not allow more than a single channel to be open to the same file at a time. To open a new channel to a file, any existing channel to that file must be closed. Additionally, depending on the implementation, output to directory files may be buffered, and, therefore, not written to disk until the channel to the file is closed. It is a good practice to keep track of open channels and to close them immediately when they are no longer required.

A channel is closed with the **close** procedure. The action,

Close "myresults".

will close the channel to **"myresults"**, allowing a new channel to be opened to that file. If the channel was open for output, then the output buffer, if one exists, is flushed and the file written to disk.

11.1.2 The Standard I/O and TTY Channels

Default I/O goes through a special channel called *the standard input channel* and *the standard output channel*, respectively. The standard I/O channels essentially provide an indirect reference to other channels.

Standard I/O is initially directed to the TTY channel (i.e., the user's terminal) but can be redirected to any other channel. The TTY channel can be referred to explicitly as **"TTY:"**; this channel is always open and cannot be closed.

Standard I/O can be redirected to a channel using the **redirect** procedure, e.g.,

Redirect input to "mydata".

opens an input channel to **"mydata"** (if one does not already exist) and redirects standard input to this channel. Similarly, executing

Redirect output to "myresults".

opens and redirects standard output to **"myresults"**. Redirecting I/O does not close the channel to which standard I/O was previously directed. If the **redirect** procedure is not given a target file device, **"TTY:"** is used by default.

Redirection of standard I/O is a temporary operation. When control returns from the ruleset in which the redirection operation was applied, the redirection is automatically undone (i.e., standard I/O is redirected back to the settings it had prior to the invocation of the ruleset). Redirection is undone even if control is returned by a nonstandard means, such as an error or user interrupt.

11.1.3 The OS Channel

Like the TTY channel, the OS channel is another special channel always open for both input and output. The OS channel, however, is open to the host operating system. Output to this channel will be executed by the host OS as though typed to it directly. Input from this channel is the results of execution that would appear on the standard output device of the OS (i.e., typically the user's terminal).

For instance, assuming the operating system is UNIX,

```
<2> Send "pwd{cr}" to "OS:".
<3> Read "{anything (bind PATH)}{cr}" from "OS:" and display PATH.
"/a/kipps/rosie/scratch"
```

we could execute the above to access the path of the current working directory.

In earlier versions of ROSIE that ran in Interlisp under TOPS-20 (Teitelman, 1978), the input/output facilities supported a special channel type called a *port*. When a port was opened, a new job was logged in on a pseudo-teletype. Text sent to the port was read by TOPS-20 as though typed by a "user," and the output of execution could be retrieved by reading from the port. Unfortunately, not all operating systems or LISPs provide the appropriate facilities for implementing ports. Thus, when ROSIE moved out of TOPS-20, ports were discarded.

The OS channel is new to ROSIE 3.0 and is an attempt to provide some of the functionality lost with ports. Unlike ports, there is only one OS channel. Also, where jobs sent to a port ran asynchronous to computations in ROSIE, jobs sent to the OS channel must run to completion before control is returned to the user's program.

The OS channel is named "OS:" and must be explicitly referenced to be used. Commands sent to the OS channel *must* be terminated with a carriage return. Also, it is an error to try to redirect standard I/O to "OS:".

NOTE: Even though the OS channel is substantially more constrained than were ports, they still may not be possible to support in some implementations of ROSIE.

11.2 THE USE OF PATTERNS

The pattern and string elements are provided primarily to support complex I/O operations. Output operations write strings to a file device. Input operations read characters from a file device, matching the characters against a pattern of "acceptable" input.

11.2.1 Sending Formatted Text

The two output procedures **send** and **print** take as an argument a string to be written to some file device. This argument can actually be any type of element. If it is not a string, it will be coerced into a string according to the following two rules:

- If a pattern element, then that pattern must describe a language of one and only one string. The pattern will be coerced into that string. These rules of coercion will be applied to the arguments of the pattern recursively. Unless the pattern explicitly specifies otherwise, the resulting string will have fixed format.
- If any other element, then the evaluation name of that element is coerced into a fixed format string. Note that if a pattern is embedded in another element, such as a tuple, it will not be treated specially from other embedded elements (i.e., its evaluation name will be coerced into a string).

As discussed in Section 9.5, strings have either a free, fixed, or mixed format. A string's format specifies the method in which the characters of the string will be displayed on the target output device. Free format strings contain no explicit line breaks; line breaks are introduced as required to make the text in the string fit the line length of the output device. Fixed format strings may contain explicit line break information; such strings are sent to the output device as is. Mixed format strings are composites of free and fixed format strings, interleaved; free format components are output to fit the line length of the device, while fixed format components are output as given.

11.2.2 Reading against a Pattern

All input takes place through the **read** procedure, which accepts as an argument a pattern against which input is matched. Characters from the input device are read one at a time until the string thus accumulated is recognized as an instance of the pattern. If at any time the pattern matcher recognizes that the string will never match the pattern, an error occurs.

Substrings of the text read from the device can only be retrieved using pattern variables. For example,

```
Read {anything (bind the reply), cr}.
```

reads a line of text terminated by an end-of-line character from standard input and binds the text (less the end-of-line character) to **the reply** (as described in Section 9.6.4) as a string. This binding is then accessible to the user's program.

An important observation to make is that the pattern matcher quits upon recognizing the shortest instance of the pattern. This means that executing

Read {anything (bind the reply)}.

will always result in binding **the reply** to the empty string (""), which is the smallest substring recognized by **anything**. ROSIE provides no "reasonableness" checker for patterns, meaning the ROSIE programmer is responsible for ensuring the correctness of all patterns used.

A final observation to note is that **read** actually will accept any type of element as an argument, not just patterns. If the element is a string, then it is coerced into a pattern describing a language consisting only of itself. If any other element type, the evaluation name of the element is coerced into a string and pattern.

11.3 CREATING TRANSCRIPT FILES

ROSIE provides a mechanism for sending everything that appears on the user's terminal to a file on disk. This is a convenient way to save a transcript of all or part of a ROSIE session. All of the example sessions appearing in this document were obtained in a *dribble session*.

After executing the **dribble** procedure, e.g.,

Dribble to "mylog".

a copy of all terminal I/O at the top-level monitor, as well as in a break monitor, is sent to *the dribble file*; a channel should never be open to the dribble file while a dribble session is active. Executing,

Stop dribbling.

discontinues the dribble operation and closes the dribble file.

Note that dribbling only saves terminal I/O when issued to a ROSIE monitor. While you may edit files, enter LISP, or jump up to the host operating system from within a dribble session, those interactions with the terminal will not be recorded in the dribble file.

11.4 INPUT/OUTPUT OPERATIONS

In the following operations: *A file* refers to a string element that identifies a text file (or an open channel) using whatever filenames conventions are appropriate for the host operating system; *a string* can be a string element or it will be coerced into a string element as described in Section 11.2.1; *a pattern* can be a pattern element or it will be coerced into a pattern as described in Section 11.2.2; *a name* can be any name element; *an integer* can be any simple number with an integer value greater than or equal to 0; and *an element* can be any element of arbitrary type. Additionally,

"TTY:" (*the TTY channel*) is a special channel open for both input and output from the terminal; by default, the standard I/O channel;

"OS:" (*the OS channel*) is a special channel open for both input and output to the host operating system.

Unless otherwise specified, operations that take a file as an optional argument will be applied to the standard I/O channel by default.

open *a file* for input
open *a file* for output

Opens a channel to *file* for the given access type. Calls an error if *file* is already open.

NOTE: A file cannot be open for both input or output, nor may more than one channel be open to the same file at the same time.

open *a file* to read
open *a file* to write

Archaic forms of **open for**. Change existing code to use **open for**.

close *a file*

Closes the channel to *file*. Calls an error if *file* is not open, or if the file is "TTY:" or "OS:".

close everything

Closes all open channels except "TTY:", "OS:" and the dribble file. Redirects standard I/O to "TTY:".

redirect input [to *a file*]
redirect output [to *a file*]

Temporarily redirects standard I/O to *file*; if no file is given, "TTY:" is used.

When executed within a ruleset, redirects standard I/O to its previous setting upon termination. In a monitor rule, resets standard I/O after executing the rule.

If a channel is not already open to *file*, one is automatically open. When this occurs, redirection to the original setting will close the channel to *file*.

NOTE: If **redirect** opens *file*, do not close *file* yourself.

NOTE: Standard I/O may not be redirected to the OS channel.

the standard input channel
the standard output channel

Produces the channel (as a string element) to which the standard I/O channels are directed.

the TTY channel

Produces "TTY:".

the OS channel

Produces "OS:".

an open channel

Produces a sequence of open channels as string elements naming the files to which they are open. These are ordered by recency (i.e., the name of the last channel open will be the first produced). Does not produce "TTY:" or "OS:".

an element is a filename

Concludes true if *element* could name a file.

NOTE: At the moment this predicate is quite primitive and will succeed if *element* is either a string or name element.

a file is open for input
a file is open for output
a file is open for input/output

Concludes true if there is an open channel to *file*, and that channel is open for input, output, or either, respectively, otherwise concludes false.

a file is available for input

Concludes true if the given file is known to the host operating system (e.g., if it can be located on disk), otherwise concludes false.

NOTE: This predicate makes the assumption that any file known to the OS can be open for input.

display an element

Outputs the evaluation name of *element* (followed by a line break) to standard input.

tab to an integer [on a file]

Causes the next character sent to *file* to be printed at the column position specified by *integer*, starting from 0.

NOTE: If the column position on the current line is already past this point, output will begin at this point on the next line.

send a string [to a file]

Outputs *string* to *file*. If applied to a nonstring, coercion to a string is automatic.

NOTE: *File* must be open for output or an error occurs.

print a string [on a file]

Like **send** with the system switch **\$PRETTYFORMAT** turned on to enhance the readability of output.

If *string* is specified using a pattern element, arguments to the pattern are coerced into strings without surrounding double or single quotes and output in lowercase, e.g.,

<11> Send {'plaintiff did suffer "a loss of one eye"', cr}.

```
'PLAINTIFF DID SUFFER "a loss of one eye"'
<12> Print {'plaintiff did suffer "a loss of one eye"', cr}.
Plaintiff did suffer a loss of one eye
```

The first letter of *string* will automatically be capitalized.

NOTE: *File* must be open for output or an error occurs.

print a name as a string

When **\$PRETTYFORMAT** is on, all instances of *name* will be output as *string*, e.g.,

```
<13> Print John Brown as "John Brown".
<14> Send {'the plaintiff did suffer "a loss of one eye"', cr}.
'JOHN BROWN DID SUFFER "a loss of one eye"'
<15> Print {'the plaintiff did suffer "a loss of one eye"', cr}.
John Brown did suffer a loss of one eye
```

read a pattern [from a file]

Reads a segment of text from *file*.

Characters are input one at a time from *file* until sufficient text has been read to either

- 1) recognize an instance of *pattern*, at which time **read** returns successfully; or
- 2) recognize that no instance can be matched, at which time **read** calls an error.

Portions of the input text can only be retrieved via pattern variables.

NOTE: *File* must be open for input or an error occurs.

For the following operations, *file* need not be open for input or output.

type a file

Lists the contents of *file* on "TTY:".

copy a file to a file

Copies the contents of the first *file* to the second. If the second *file* already exists, its old contents are destroyed.

append *a file* to *a file*

Appends the contents of the first *file* to the end of the second.

rename *a file* to *a file*

Changes the name of the first *file* to the second. If a file by that name already exists, it is destroyed.

delete *a file*

Deletes *file* from the user's directory. Does not ask for conformation. No error occurs if *file* does not actually exist.

dskin *a file*

Loads *file* using the implementation LISP's **load** function or its equivalent. Provided for loading LISP files into the system.

dribble to *a file*

Opens a special output channel to *file*, making it the dribble file. After this, a copy of all terminal I/O will be sent to *file*. The dribble file may not be closed except with **stop dribbling**.

NOTE: You may edit files while dribbling, but that part of the session will *not* be dribbled.

stop dribbling

Closes the dribble file and stops copying terminal I/O. If no dribble file is open, an error occurs.

XII. ERRORS AND ERROR RECOVERY

Runtime errors are either recoverable or nonrecoverable, and either system generated or generated by the user's program. When a runtime error occurs, execution is temporarily suspended, and an error message is printed indicating the problem and the ruleset in which it was encountered. While ROSIE does not support an elaborate error handler, it does provide a mechanism for trapping and recovering from runtime errors.

12.1 NONRECOVERABLE AND USER ERRORS

When one of the few nonrecoverable errors occurs or when the user's program calls an error, control is unconditionally returned to the top-level monitor and all computations aborted. A user's program can call an error, aborting computations, with the **quit** procedure, i.e.,

quit [*because a string*]

Throws control to the top-level monitor. If the **because** option is given, the string is printed to the standard output channel (normally the user's terminal) before aborting computation.

12.2 RECOVERABLE ERRORS

Most runtime errors in ROSIE 3.0 are recoverable. This means that they can be trapped and possibly fixed, allowing computations to resume from the offending rule gracefully. In addition, user interrupts, signaled by hitting <ctrl>C, are treated as recoverable errors.

When a recoverable error occurs, ROSIE does two things in sequence. First, it attempts to invoke an assert demon that, if defined, traps the error and permits automatic error recovery. If no such demon exists, or if it did not signal for computations to be resumed, control is thrown into a break loop. Within the break loop, the user can edit the offending rule, fix the error, and resume computations.

For further information on interactions in the break loop, see Section 14.3.

12.3 THE ERROR DEMON

Automatic error recovery can be controlled through the use of a special assert demon called *the error demon*. When processing a recoverable error, ROSIE simulates an assertion of the proposition

<string, filesegment> is an error

where *<string, filesegment>* is a tuple element containing *string*, which identifies the error message, and *filesegment*, which identifies the ruleset rule causing the error.

This proposition is not actually asserted into the database, but it will invoke an assert demon of the form

Before asserting a message is an error:

if such a demon exists. Further, if the error demon executes the **continue** procedure, then computation will be resumed automatically at the point of the error call.

XIII. THE FILE PACKAGE

The file package is the heart of ROSIE's programming environment. It helps the user build, modify, examine, and maintain programs in a way that exploits the modular and English-like nature of ROSIE rulesets. It also encourages interactive and real-time system development by minimizing the parsing and compiling overhead caused by changes to rules and rulesets.

13.1 PROGRAM FILES

A ROSIE program is developed and maintained in a *program file*. A program file is actually a set of files that resides in the user's directory. These files contain the source code of the program, the code derived from parsing the program, and the binary code representation of the program derived by further compilation.

The component files of the program file share a common root name (i.e., the name given the program file when created with the **build** procedure). They are distinguished by their extension. These files are explained briefly below:

- *file.txt* -- contains the actual source code of the program in its original text form. It is the only "human" readable file in the group and is the file that should be printed when a hardcopy of the program is required.
- *file.map* -- contains the executable *HILEV* representation of the code in the *.txt* file, as well as a map linking the corresponding components of both files. The contents of the *.map* file are what is actually loaded and modified during program development.
- *file.cmp* -- contains the binary code representation of a program compiled from the *HILEV* in the *.map* file. This file is created by the **compile** procedure and normally exists only when the user is satisfied with the behavior of a system and desires to improve performance.

None of these files should ever be modified directly by the user since each is required by the file package and is related in unobvious ways.

A program file is named by a string element that is consistent with the file-naming conventions of the host operating system. ROSIE is not extremely sophisticated about file-naming conventions and may do the unexpected when accessing files in other than the current working directory. It is hoped that such issues will be addressed at some future date.

13.2 USING THE FILE PACKAGE

To the user, a program file is simply a text file containing ROSIE source code. A program file is created with the **build** procedure. This procedure takes a single argument that becomes the name of the program file. Once created, the user can add code to the empty program file, and, when finished, write it to disk.

To work on an existing program file, the user must first *notice* it. This is normally done with the **load** procedure, which brings the **.map** file into the system, noticing the contents of that file as well as *enabling* its rulesets and executing its *file rules*.¹

The user can examine the contents of a program file with the **list** procedure, which shows the program's source code drawn from the **.txt** file. The **scan** procedure also lists the contents of a program file, but in an abbreviated form. The user can examine ROSIE's interpretation of his code with the **deparse** procedure. The deparser automatically generates source code from the HILEV representation of a program, illustrating ROSIE's interpretation of the program with indentation and parentheses.

The user may change portions of a program file with the **copy**, **move**, and **erase** procedures. The **edit** procedure allows a user to edit noticed program files, or portions of such files, while **insert** allows a user to insert edited code before or after some portion of a noticed program file. After editing, changes to a program file can be written to disk with the **save** procedure.

When the system builder is satisfied with the behavior of his code, ROSIE permits him to optimize his program with the **compile** procedure. **Compile** converts ruleset definitions and file rules into binary code, storing this code in the **.cmp** file. Loading a compiled program file will enable the compiled definitions and execute the compiled file rules.

The **sysload** procedure can be used to load a file without noticing it. This is more efficient, but does not allow the user to edit or examine the contents of the file. Similarly, the **notice** procedure can be used to notice a file without actually enabling it, while the **enable** procedure enables the contents of a noticed program file.² There is also a **disable** procedure, which undefines rulesets, an **unnotice** procedure, which causes the file package to forget what it knows about a file, and an **unload** procedure, which first disables a file and then unnotifies it.

¹Any rule that is not part of a ruleset is a file rule.

²Enabling defines the rulesets and executes the file rules of the program file.

Finally, the **parse** procedure permits users to build program files for code that was edited or developed outside of the ROSIE environment. For instance, the **.text** files of pre-ROSIE 3.0 programs *must* be ported to ROSIE 3.0 using **parse**.

13.3 DEFINING RULESETS AND FILE RULES

A program file may contain any number of ruleset definitions or file rules. When loaded, the contents of the program file are noticed and enabled. Enabling defines the program file's rulesets, allowing them to be invoked. Enabling also executes the program file's file rules.

Actually, when a program file is loaded, only its rulesets without syntax errors are noticed and enabled. Rulesets that contain syntax errors in anything other than their header are not enabled but are still noticed. Rulesets that contain syntax errors in their header are not even noticed and may be fixed only by editing the entire program file. File rules are collected in a group to be executed after all rulesets in the file have been enabled and the file closed. If the file contains any syntax errors, then none of its file rules are executed.

As an illustrated example, consider the following sample session with the program file, called **"integers"**, which prints a sequence of integers from 1 to 5. It contains two file rules and one ruleset definition.

```
(R)
[ ROSIE      Version 3.0  (PSL)  26-May-86 ]

<2> Load "integers".

Loading 'FILE: "integers"'...
TO PRINT-NUMBERS
Done loading.

<3> List "integers".

[rule 1] Let the first number be 1.

[rule 2] Let the last number be 5.

To print-numbers:
  [ This rule prints a sequence of numbers. ]
[1] For each integer from the first number to the last number,
    display that integer.
End.

<4> List 'file: "integers", to print-numbers, 1'.
```

```
[ This rule prints a sequence of numbers. ]  
[1] For each integer from the first number to the last number,  
    display that integer.
```

```
<5> Print-numbers.
```

```
1  
2  
3  
4  
5
```

In the example above, note the rule numbers printed as comments at the beginning of every file rule and ruleset rule. These comments are inserted and updated automatically by the file package. They are displayed along with the rule when examining or editing the rule text. Individual rules may be cited by number (e.g., statement <4> above demonstrates the use of a filesegment to examine the text of the first rule of the ruleset) and, once identified, may be passed as arguments to any of the file package operations.

Comments that appear in a program file are always associated with the closest file item after the comment. When such an item is examined or edited (as in statement <2>), the text of the comments appears with the text of the item.

A recommended method for organizing large programs is to maintain three separate program files: one for regular (nonsystem) rulesets, another for system rulesets, and a third for file rules. Since system rulesets and file rules are normally simple to debug, this scheme allows the user to compile these two components of a system early and concentrate on the task of developing the main body of code.

13.4 EDITING AND MODIFYING PROGRAM FILES

Some file package operations, such as **edit**, **insert**, **copy**, and **move**, are used to modify the text of program files. In the past, this aspect of the file package often became a bottleneck in the rapid development of a ROSIE program. In ROSIE 3.0, these processes have been substantially revised with the goal of improving the speed with which editing tasks can be performed.

In earlier releases of ROSIE, any changes to a program file were immediately written to disk. This was done to ensure that edits were not lost. While a fine idea, in practice it restricted program development considerably by making even the smallest edits excessively time consuming. In addition, any program text sent to the editor was unconditionally reparsed, regardless of what changes were actually made to it. While filesegments allowed users to edit small portions of a file, files that needed editing in several spots required several calls to the editor, each one again writing the program file to disk.

In 3.0 release of ROSIE, all changes to a program file are maintained in core. To update the `.txt` and `.map` files, the user must explicitly call the `save` procedure.³ In addition, rules and other file items sent to the editor will be reparsed only if their post-edit text is detectably different from their pre-edit text.⁴ These two changes to the file package, as well as others not mentioned, greatly enhance the speed with which edits can be made.

NOTE: The approach taken by ROSIE 3.0 requires considerably more internal storage than the older approach. This increases the amount of time spent doing garbage collection and other memory management tasks. Users are advised to save edits whenever convenient, compiling files when satisfied with their behavior, and only noticing files that are to be edited or debugged.

13.5 USING FILESEGMENTS

```
<filesegment> ::= ' FILE : <term> [, <header>] [, <rule spec>] '
               ::= ' <header> [, <rule spec>] '
```

```
<rule spec>   ::= <integer> [<integer>]
               ::= BEFORE <term>
               ::= AT <term>
               ::= FROM <term> TO <term>
               ::= AFTER <term>
```

The filesegment element (described in Section 9.7) provides the ability to identify program files, rulesets, or sequences of file rules or ruleset rules. Each file package operation requires a filesegment argument. Many file package operations work on portions of a file, such as a ruleset or individual rules, as well as the entire program file. To facilitate ease of use, filesegments may be specified in a convenient shorthand notation.

The following are examples of filesegments for identifying components of the "integers" program file. The filesegment

```
'file: "integers"'
```

identifies the entire program file. The filesegment

³If edited but unsaved files are detected when exiting a ROSIE session, the user will be informed of their existence and asked whether these files should be saved.

⁴Essentially, this means that if the text of a rule is not changed, it will not be reparsed.

`'file: "integers", 1'`

specifies the first file rule of that file, while

`'file: "integers", 1 2'`

specifies every file item between the first and second file rules. The filesegment

`'file: "integers", to print-numbers'`

identifies the one ruleset, and the filesegment

`'file: "integers", to print-numbers, 1'`

identifies the first rule of that ruleset.

It is also possible to identify a ruleset without specifying the program file from which it originated, e.g.,

`'to print-numbers'`

If there exists an enabled ruleset (noticed or not) with a matching header, then it is the ruleset identified by the filesegment. If no such ruleset is enabled, ROSIE searches the list of noticed rulesets. The filesegment identifies the first of these to match. If this fails, then the filesegment is considered to be unknown to the file package and an error occurs.

13.5.1 Rule Sequence Specifiers

As seen above in the BNF for filesegments, file rules and ruleset rules may be identified by number using a rule sequence specifier--rule numbers must be positive integers. There are two variations on rule sequence specifiers.

The simplest form,

`<rule spec> ::= <integer> [<integer>]`

is a shorthand intended for use at the top-level monitor. It accepts only integers (as opposed to arbitrary terms) as rule specifiers. When specified as

`. . . , 3'`

it identifies the third rule, while

`. . . , 1 4'`

identifies the first through the fourth rule, inclusive.

The other form,

```
<rule spec> ::= BEFORE <term>
              ::= AT <term>
              ::= FROM <term> TO <term>
              ::= AFTER <term>
```

permits arbitrary terms (which must evaluate to positive integers) to specify rules. This form offers greater functionality and readability than the other. When specified as

... , before 4'

it identifies either every file item from the beginning of the program file up to, but not including, the fourth file rule, or every item of a ruleset between its header and fourth rule, exclusive. When specified as

... , at 3'

it identifies the third rule only;

... , from 3 to 7'

identifies the third through seventh rule, inclusive. Finally,

... , after 4'

either identifies every file item immediately after the fourth file rule, or every ruleset rule between the fourth rule and the end statement, exclusive.

13.5.2 Shorthand Notation

The examples above illustrate the formal syntax of filesegments. There is also a convenient shorthand notation that is recognized by all of the file package operations. While this shorthand excludes the specification of rule sequences, it does allow files and rulesets to be identified simply and easily.

The following are examples of this shorthand for naming elements in the "integers" program file:

Shorthand	Filesegment
"integers"	'file: "integers"'
PRINT-NUMBERS	'file: "integers", to print-numbers'
PRINT	"
NUMBERS	"
RIN	"

The shorthand string names either a loaded (and noticed) program file or a program file that exists on disk. If no such program file can be found an error occurs.

A name element indicates a noticed ruleset. The element is compared to the name of each noticed ruleset, using the following rules:

- If the element matches the name of one and only one ruleset exactly, then that ruleset is selected.
- If it exactly matches more than one ruleset, then the user is queried to choose the correct ruleset.
- If the characters in the element exactly match some substring of the name of one and only one ruleset, then that ruleset is selected.
- If it partially matches more than one ruleset, the user is queried to choose the correct one.

If the name cannot be found to identify a ruleset, then it is coerced into a lowercase string and treated as the shorthand for a file. If no such file exists, an error occurs.

13.6 FILE PACKAGE OPERATIONS

In the following operations, a *filespec* refers to either a filesegment element or the shorthand notation for a filesegment--only the contents of noticed program files can be designated in shorthand--and a *file* refers to a string element that names a text file.

build a *filespec*

Creates a new program file for *filespec*. **Build** leaves the file noticed, so the user can immediately begin editing it. The file must be saved before it appears on disk.

load a *filespec*

Loads and notices a program file or a portion of a program file.

If *filespec* names a program file, e.g.,

load 'file: "myprog"'

the entire file is loaded into the ROSIE session. Rulesets defined in the file will be defined in ROSIE. If the file contains no syntax errors, then its file rules will be executed after its rulesets have been defined.

If the **.cmp** file exists and if the write date of the **.cmp** file is more recent than that of the **.map** file, the compiled version of the program file will be loaded.

If *filespec* names a sequence of file rules or a ruleset from a file, e.g.,

load 'file: "myprog", to move a ship from a port'

then the entire program file is noticed, but only the specified portion of the file is enabled.

NOTE: When a portion of a program file is specified, it will always come from the **.map** file and never from the **.cmp** file.

sysload a filespec

Same as **load**, but the file is not noticed and so cannot be listed or edited. Sysloaded files require less memory.

Program files that are not going to be edited or examined should be sysloaded--for example, the *system ruleset library* is sysloaded. Noticing a file requires a large amount of space, and too many large noticed files will significantly impair system performance.

Unlike earlier releases, in ROSIE 3.0 sysloaded rulesets may be broken, traced, and disabled (although not re-enabled).

notice a filespec

Notifies the contents of a program file without enabling the file's rulesets or executing its file rules. Rulesets can later be defined and file rules executed with the **enable** procedure, i.e., the **load** procedure is essentially a **notice** followed by an **enable**.

enable a filespec

Enables the given filesegment. If this is a program file, defines its rulesets and executes its file rules. If a ruleset, simply defines that ruleset. If the filesegment is not already noticed, attempts to load the appropriate program file.

compile a filespec

Compiles a program file, storing the binary code in a **.cmp** file.

If the file is not already noticed, it will be noticed automatically.

After compilation, the *.cmp* file will be loaded.

A program file can be compiled even if portions of it contain syntax errors. A ruleset must be free of syntax errors before it can be compiled.

If *filespec* specifies a ruleset, then only that ruleset will be compiled. The resulting binary will not be sent to disk, but it will replace the ruleset's definition in core.

change a filespec to a filespec

Renames the program file designated by the first *filespec* to the program file named by the second. This is the only safe way to rename a program file.

parse a file

Converts *file* (assumed to be a text file containing the source code to a ROSIE program) into a program file. The new program file is automatically loaded.

list a filespec

Displays the text associated with *filespec* to the terminal.

scan a filespec

Summarizes the contents and status of *filespec*, which indicate syntax errors and other relevant information, e.g.,

```
<6> Scan "integers".
```

```
'FILE: "integers"' contains:  
  [Rules 1 and 2]  
  TO PRINT-NUMBERS [1 rule]
```

deparse a filespec

Lists ROSIE's interpretation of *filespec* by "deparsing" its associated HILEV code.

The deparser is essentially a text generator, i.e., given a piece of HILEV code, it produces the ROSIE source code equivalent. This machine-generated source code illustrates how terms and clauses were interpreted with proper indentation and by going to the extreme in

its use of parentheses as delimiters, e.g.,

<7> Deparse "integers".

[Rule 1] LET THE FIRST NUMBER BE 1.

[Rule 2] LET THE LAST NUMBER BE 5.

TO PRINT-NUMBERS:

[1] FOR EACH INTEGER (FROM THE FIRST NUMBER) (TO THE
LAST NUMBER),
DISPLAY THAT INTEGER.

END.

This operation can be essential to understanding and debugging complex ROSIE expressions.

decode *a filespec*

Lists the HILEV code associated with *filespec*.

erase *a filespec*

Removes *filespec* from its program file. Erasing an entire program file does not delete the *.txt* and *.map* files, but merely nulls out their contents. In ROSIE 3.0, **erase** no longer asks for confirmation from the user before making the change.

NOTE: Erasing an enabled ruleset does not disable the ruleset. For this, one must use the **disable** procedure.

copy *a filespec* **before** *a filespec*
copy *a filespec* **after** *a filespec*

Copies (splices) the first *filespec* before or after the second.

move *a filespec* **before** *a filespec*
move *a filespec* **after** *a filespec*

Equivalent to a **copy** followed by an **erase** of the first *filespec*.

NOTE: Some intuitive restrictions are placed on the movements of particular types of filesegments. For instance, although it is permissible to copy the rules of one ruleset into another ruleset, one may not copy an entire ruleset into another.

edit *a filespec*

Invokes the user's text editor (see below) to edit the text of *filespec*. The edited filesegment is then parsed and loaded, replacing the original. ROSIE allows users to edit entire program files, sequences of file rules, rulesets, or sequences of ruleset rules (i.e., any filesegment).

insert before *a filespec***insert after** *a filespec*

Invokes the user's text editor, allowing the user to compose code that should be inserted before or after *filespec*.

NOTE: The "user's editor" is taken from the lisp variable **\$ROSIEEDITOR**, which will be bound from the environment variable **EDITOR** if possible or to "edit" if not. The user may specify a text editor of preference by setting the **EDITOR** environment variable at the OS level or by setting the **\$ROSIEEDITOR** lisp variable in the **.rosierc** file, e.g.,

```
(SETQ $ROSIEEDITOR "emacs")
```

NOTE: The **erase**, **copy**, **move**, **edit**, and **insert** operations do *not* automatically write program file changes to disk. This must be done explicitly using the **save** operation.

save [*a filespec*]

Updates the **.txt** and **.map** files of *filespec* on disk to reflect changes made during a ROSIE session.

If *filespec* is not given, ROSIE appraises the user of all program files that have been modified and need to be saved, giving the user the option of saving each in turn.

NOTE: **Save** is called by **logout**.

disable *a filespec*

The inverse of the **enable** procedure. If *filespec* names a program file, **disable** undefines all rulesets of that file; if a ruleset, disables only that ruleset.

NOTE: A ruleset need not be noticed to be disabled, but it may not be re-enabled without being noticed.

unnotice *a filespec*

The inverse of the **notice** procedure. If *filespec* names a program file, **unnotice** causes the file package to forget about everything in that file (this is *not* equivalent to **erase**); if a ruleset, causes the file package to forget about just that ruleset.

unload *a filespec*

The inverse of **load**; a **disable** followed by an **unnotice**.

find *a string in a filespec*

Lists the component filesegments in *filespec* in whose text appears *string*. Upper- and lowercase characters are treated equivalently.

When an instance of *string* is found, lists the filesegment in which it appears as well as the first line that references *string*, e.g.,

```
<8> find "number" in "integers".
```

```
Searching 'FILE: "integers"...
```

```
'FILE: "integers", AT 1'
```

```
  "Let the first number be 1."
```

```
'FILE: "integers", AT 2'
```

```
  "Let the last number be 5."
```

```
'TO PRINT-NUMBERS'
```

```
  "To print-numbers:"
```

```
'TO PRINT-NUMBERS, AT 1'
```

```
  "For each integer from the first number to the last number,"
```

```
Done searching.
```

XIV. THE BREAK PACKAGE: DEBUGGING PROGRAMS

The break package is a facility for monitoring and debugging programs. It is primarily intended for use with rulesets and ruleset rules, but it can be used to monitor other aspects of a program as well. It includes features that allow a user to resume from recoverable errors, to monitor control flow, to interrupt and examine ruleset invocations as well as the results they generate, and to profile the effective computation time of various aspects of a ROSIE program.

The three components of the break package, the *trace*, *break*, and *profile* facilities, allow the user to temporarily modify or *break* selected ruleset or demon definitions (even if the selected rulesets or demons are not defined) and access different features of the break package.

14.1 BREAKABLE ASPECTS OF A PROGRAM

Any aspect of a program that is capable of invoking a ruleset or demon is breakable, even if no such ruleset or demon exists. Thus, it is possible to break database actions, such as assertions, denials, and tests of a proposition, and generation from a class, in addition to calls on defined rulesets and demons.

To trace assertions of a particular proposition, one would break the assert demon for that proposition, e.g.,

Trace 'before asserting a man does love a woman'.

Similarly, to trace the values produced for some class, one would break the produce demon for that class, e.g.,

Trace 'before producing a target at an airfield'.

Both actions break the demons that would normally be invoked before the occurrence of the particular assert or produce event. If these demons are not defined, then a dummy (no-op) definition will be enabled and broken.

It is likewise possible to break undefined rulesets. When a break throws control into the interactive *break loop*, monitor-level commands are executed in the context of the ruleset invocation (i.e., as though they are actually rules of the ruleset). Thus, breaking undefined rulesets allows the user to play the part of the ruleset definition. This offers an appealing aid to program development where the user is uncertain of how particular rulesets should behave and wishes to experiment. If an undefined ruleset that is broken becomes defined, the dummy definition is replaced by the new definition, however the ruleset remains broken. Rulesets may be unbroken only by explicit command.

Within defined rulesets and demons (except system rulesets), it is also possible to break the execution of individual rule. For instance,

Break 'to move a ship to a destination, 3'.

will cause control to enter a break loop prior to executing the third rule of the **move to** procedural ruleset. Note however that redefining the ruleset (e.g., after an edit) will remove the break.

Finally, when the ruleset being broken is either a generator ruleset or a generate demon, the produce demon for its class is also broken automatically, since the values being produced are only visible from the produce demon. When either the generator ruleset or generate demon are unbroken, the produce demon will be unbroken as well.

14.2 THE TRACE FACILITY

The trace facility modifies the definition of a ruleset to display a message whenever control passes in or out of the ruleset. Upon invoking a traced ruleset, a message is printed stating the trace depth and the title of the broken ruleset, with formal parameters replaced by the values of actual parameters. When exited, another message is printed stating that control is returning from the ruleset. Results, if any, are displayed at this time as well.

The results of a ruleset are dependent upon the ruleset's type. Procedural and generator rulesets never return a value.¹ Predicates can return a conclusion of true or false or make no conclusion at all. Demons can either continue the interrupted event or not. Tracing a ruleset rule has the same effect as described above, with the additional effect of displaying a message prior to executing the rule.

The following example illustrates a trace of a demo program called "players".

<13> List "players".

To find basketball players:

[1] Send "{Every man who is tall} is a basketball player.{cr}".
End.

To decide if a person is tall:

[1] If the person's height is greater than 6.7 feet,
conclude true, otherwise conclude false.
End.

¹The values produced by a generator are bound to its description variable and not explicitly returned.

To generate a man:
 [1] Produce each of Jim, Jack and John.
 End.

To generate the height of a person:
 [1] Select the person:
 <Jim> produce 6.4 feet;
 <Jack> produce 6.9 feet;
 <John> produce 5.8 feet.
 End.

<14> Trace "players".

Breaking 'FILE: "players"'...
 TO FIND BASKETBALL PLAYERS -- broken.
 TO DECIDE IF A PERSON IS TALL -- broken.
 TO GENERATE A MAN -- broken.
 BEFORE PRODUCING A MAN -- broken.
 TO GENERATE THE HEIGHT OF A PERSON -- broken.
 BEFORE PRODUCING A HEIGHT OF A PERSON -- broken.
 Done breaking.

<15> Find basketball players.
 1: TO FIND BASKETBALL PLAYERS
 2: GENERATING A MAN
 2: Producing JIM
 3: TESTING IF JIM IS TALL
 4: GENERATING A HEIGHT OF JIM
 4: Producing 6.4 FEET
 3: Concluding FALSE from:
 TESTING IF JIM IS TALL
 2: Producing JACK
 3: TESTING IF JACK IS TALL
 4: GENERATING A HEIGHT OF JACK
 4: Producing 6.9 FEET
 3: Concluding TRUE from:
 TESTING IF JACK IS TALL
 JACK is a basketball player.
 2: Producing JOHN
 3: TESTING IF JOHN IS TALL
 4: GENERATING A HEIGHT OF JOHN
 4: Producing 5.8 FEET
 3: Concluding FALSE from:
 TESTING IF JOHN IS TALL
 1: Returning from:
 TO FIND BASKETBALL PLAYERS
 <16> Untrace "players".

Unbreaking 'FILE: "players"'...
 TO FIND BASKETBALL PLAYERS -- redefined.

```
TO DECIDE IF A PERSON IS TALL -- redefined.  
BEFORE PRODUCING A MAN -- disabled.  
TO GENERATE A MAN -- redefined.  
BEFORE PRODUCING A HEIGHT OF A PERSON -- disabled.  
TO GENERATE THE HEIGHT OF A PERSON -- redefined.  
Done unbreaking.
```

Note above that tracing a generator also traces its associated produce demon, e.g., as in the case of the generators for **man** and **height**. A result of this can be observed in the generation of each **man** at level 2 of the trace. The first comment at level 2 comes from the generator, stating simply that it is beginning to generate from the class of **man**. All other comments at level 2 come from the associated produce demon, stating the elements that are being produced for this class.

14.3 THE BREAK FACILITY

The break facility modifies a ruleset definition to halt the invocation of that ruleset temporarily and pass control to an interactive monitor called a *break loop*. Commands at the break loop level are executed within the context of the ruleset invocation. From within a break loop, the user can examine and change the private database of the broken invocation, or move down the stack of ruleset invocations, examining and changing the private databases at other levels.

When the broken ruleset is a generator, two levels of break occur; the first as the ruleset is invoked and the second whenever the ruleset attempts to produce an element. This gives the user the ability to examine and modify each element produced.

14.3.1 Break Commands

Control can enter a break loop when either a broken ruleset is invoked or when a continuable runtime error occurs (see Chapter 12). Within a break loop, the user can interrogate the system, perform normal computations, examine the context of the invocation, and continue or return from the ruleset. Additionally, the break loop gives the user access to a number of special *break commands*.

The break commands cannot be invoked in conjunction with other actions. They must appear as singleton commands to the monitor. All break actions end with an exclamation point (!); this punctuation is used to avoid confusion with procedures of the same name.

The break commands are available from within a break loop:

eval!

Resumes computation without releasing control of the broken ruleset or rule. Once the computation is completed, control is returned to the break monitor, where the user can examine the results of the computation or its effect on the system state. When the break is finally released, computation of the broken ruleset or rule will not be unnecessarily repeated.

NOTE: This command is not operational when the break loop was entered due to a runtime error.

result!

Displays the results (if any) from the evaluation of the broken ruleset or rule.

list [(| ruleset | <integer>)]!

Calls the file package operation **list** on the broken ruleset or ruleset rule. If no argument is given, calls **list** on the broken filesegment. If the **ruleset** option is given, calls **list** on the entire ruleset. If the <integer> option is given, calls **list** on that rule of the broken ruleset.

edit [(| ruleset | <integer>)]!

Like **list**, except that it applies the file package operation **edit** to the designated filesegment.

resume [(| ruleset | <integer>)]!

Releases the break and allows computation of the broken ruleset or rule to resume. If this computation was previously made via the **eval** command, then the evaluation of the ruleset is not repeated.

The **ruleset** and <integer> options are operational only if the break loop was entered by a runtime error. In such cases, these options allow you to restart the ruleset invocation or resume computations from any rule in the ruleset. If no argument is given, resumes computation from the rule that triggered the error.

return!

Releases the break, throwing control out of the ruleset invocation without allowing computation of the broken ruleset to continue.

conclude true!
conclude false!

Releases the break, concluding true or false without resuming computation of the broken ruleset.

NOTE: May be used only within a broken predicate ruleset.

produce an element!

If called from within a generator ruleset, attempts to produce *element*, possibly invoking a produce demon. If called from within a produce demon, actually replaces the element currently being produced with *element*.

The difference between the two cases is subtle but distinct. In the first case, *element* is produced as the one and only element generated by the ruleset. In the second, it is produced as just one of the elements generated. In either case, the break is released without computation of the broken ruleset being resumed.

NOTE: May be used only within a generator ruleset or a produce demon.

trace!

Displays a backtrace of suspended ruleset invocations.

Each invocation is depicted using the ruleset's header followed by the number of the rule currently being invoked. A star (*) is placed to the left of the current (i.e., examinable) ruleset frame.

The current frame at the time of the break is a frame of the broken ruleset. This is also the frame at the top of the stack. The frame at the very bottom (which is not accessible) is the frame for the top-level monitor.

down!

Changes the current frame to the next frame down on the frame stack, and thus allows the examination of the calling ruleset's private database.

up!

Changes the current frame to be the next higher frame on the frame stack.

top!

Sets the current frame to its original value (i.e., the frame of the broken ruleset).

bottom!

Sets the current frame to be the frame for the ruleset invocation at the bottom of the stack (i.e., the frame of the initial ruleset invocation).

quit!

Throws control back to the last break point. That is to say, if the user is nested several layers deep in break monitors, control can be returned to earlier layers successively.

pop!

Throws control back to the ruleset invocation that called the currently broken ruleset. That is, if ruleset X invoked ruleset Y and a break was called from ruleset Y, **pop!** would forget about the invocation of Y and make it appear as though the break was called from the invocation of X. This command provides a means resuming computation from any point on the frame stack.

help!

Displays a short synopsis of the break actions.

In addition to these, the following operations can also be useful from within a break:

private?

Displays the contents of the private database in the current frame.

produce *an element*.

When used within a generator ruleset, attempts to produce *element* without releasing the break afterwards.

quit.

Aborts computation, throwing control back to the top-level monitor.

14.3.2 Example Session

In the following example of break loop interactions, the test rulesets are taken from the "players" program seen earlier.

```
<18> Break each of tall and height.
```

```
Breaking 'FILE: "players"'...
TO DECIDE IF A PERSON IS TALL -- broken.
Done breaking.
```

```
Breaking 'FILE: "players"'...
TO GENERATE THE HEIGHT OF A PERSON -- broken.
BEFORE PRODUCING A HEIGHT OF A PERSON -- broken.
Done breaking.
```

```
<19> Find basketball players.
```

```
Broken at:
  'TO DECIDE IF A PERSON IS TALL'.
```

```
[1] Private?
[ PRIVATE Database ]
  JIM IS A PERSON.
```

At this point, control has just entered the break loop. The first thing we do is check the broken ruleset's private database to see how the predicate is being applied.

```
[2] List!
```

```
> [1] If the person's height is greater than 6.7 feet,
>       conclude true, otherwise conclude false.
```

```
[3] List ruleset!
```

```
> To decide if a person is tall:
>
> [1] If the person's height is greater than 6.7 feet,
>       conclude true, otherwise conclude false.
>
> End.
```

In line [2], we examine the break point; not surprisingly, this is the first rule. In line [3] we examine the entire ruleset.²

²Note that some output is offset by angle brackets (>). This convention is used here to avoid confusion with type-in and will not appear in an actual ROSIE session.

[4] Eval!

Broken at:

'TO GENERATE THE HEIGHT OF A PERSON'.

[1] private?

[PRIVATE Database]
JIM IS A PERSON.

[2] List!

> [1] Select the person:
> <Jim> produce 6.4 feet;
> <Jack> produce 6.9 feet;
> <John> produce 5.8 feet.

[3] Resume!

The **eval!** command in line [4] allows computation to resume without releasing control of the break. Control then enters another level of break when the broken **height** generator is invoked. In line [1] and [2], we examine the private database and the break point, respectively. The **resume!** command in line [3] allows computation to continue and *releases* the control of the break.

Broken at:

'BEFORE PRODUCING A HEIGHT OF A PERSON'.

Tentatively producing:
6.4 FEET

[1] private?

[PRIVATE Database]
JIM IS A PERSON.
6.4 FEET IS A HEIGHT.

Control is again thrown into a break loop when the broken produce demon associated with the **height** generator is invoked. Notice that when we examine the private database of this ruleset we find an extra assertion; this is the value being produced.

[3] Resume!

Continuing from:

'BEFORE PRODUCING A HEIGHT OF A PERSON'

Returning from:

'TO GENERATE THE HEIGHT OF A PERSON'

Evaluated.

[5] Result!

Concluding FALSE.

When we resume computations, control returns to the top-most break loop, where we can examine the results of the computation. Now, say we don't like these results. We could do the following.

[6] Eval!

Broken at:

'TO GENERATE THE HEIGHT OF A PERSON'.

[1] Resume!

Broken at:

'BEFORE PRODUCING A HEIGHT OF A PERSON'.

Tentatively producing:

6.4 FEET

[1] Produce 6.8 feet!

Producing 6.8 FEET for:

'BEFORE PRODUCING A HEIGHT OF A PERSON'

Returning from:

'TO GENERATE THE HEIGHT OF A PERSON'

Evaluated.

[7] Result!

Concluding TRUE.

[8] Resume!

Concluding TRUE for:

'TO DECIDE IF A PERSON IS TALL'

JIM is a basketball player.

At line [6], we reevaluated the computation, continuing as before. At line [1] of the produce demon break loop, however, instead of allowing the tentative value to be produced, we instead produce a different value. This value passes the test for tallness, causing the predicate to succeed. We now release the break, and the **find** procedure outputs a message about a candidate basketball player.

Broken at:

'TO DECIDE IF A PERSON IS TALL'.

[1] Private?

[PRIVATE Database]

JACK IS A PERSON.

[2] Resume!

Broken at:

'TO GENERATE THE HEIGHT OF A PERSON'.

[1] Resume!

Broken at:

'BEFORE PRODUCING A HEIGHT OF A PERSON'.

Tentatively producing:

6.9 FEET

Control enters a break loop again on Jack. Here we continued computations up to the produce demon.

[1] Trace!

* BEFORE PRODUCING A HEIGHT OF A PERSON
 TO GENERATE THE HEIGHT OF A PERSON [rule 1]
 TO DECIDE IF A PERSON IS TALL [rule 1]
 TO GENERATE A MAN [rule 1]
 TO FIND BASKETBALL PLAYERS [rule 1]
 [Top-level Monitor]

[2] Down!

* TO GENERATE THE HEIGHT OF A PERSON

[3] Down!

* TO DECIDE IF A PERSON IS TALL

[4] Down!

* TO GENERATE A MAN

[5] Trace!

BEFORE PRODUCING A HEIGHT OF A PERSON
 TO GENERATE THE HEIGHT OF A PERSON [rule 1]
 TO DECIDE IF A PERSON IS TALL [rule 1]
 * TO GENERATE A MAN [rule 1]
 TO FIND BASKETBALL PLAYERS [rule 1]
 [Top-level Monitor]

[6] private?

[PRIVATE Database]

[7] List ruleset!

> To generate a man:

>

> [1] Produce each of Jim, Jack and John.

>

> End.

[8] Top!

* BEFORE PRODUCING A HEIGHT OF A PERSON

[10] Pop!

Broken at:

'TO GENERATE THE HEIGHT OF A PERSON'.

[2] Pop!

Broken at:
 'TO DECIDE IF A PERSON IS TALL'.

The interactions seen above demonstrate how one can examine the computation environment and traverse the stack of ruleset invocations. The **trace!** command given in line [1] prints the invocation stack--most recently invoked first. Lines [2] thru [4] show how to move down to the stack so that in lines [5] thru [7] it looks like control is in the generator for **man**. In line [8], we jump back up to the broken demon. In lines [10] and [2], we pop invocations off the top of the stack. This allows us to resume computations from any point on the stack, as seen below.

[3] Conclude false!
 Concluding FALSE for:
 'TO DECIDE IF A PERSON IS TALL'

Broken at:
 'TO DECIDE IF A PERSON IS TALL'.

[1] Private?
 [PRIVATE Database]
 JOHN IS A PERSON.

[2] Conclude true!
 Concluding TRUE for:
 'TO DECIDE IF A PERSON IS TALL'
 JOHN is a basketball player.
 <20> unbreak.

Unbreaking...
 BEFORE PRODUCING A HEIGHT OF A PERSON -- disabled.
 TO GENERATE THE HEIGHT OF A PERSON -- redefined.
 TO DECIDE IF A PERSON IS TALL -- redefined.
 Done unbreaking.

14.4 THE PROFILE FACILITY

The profile facility modifies ruleset definitions so that they gather statistical data on the computational costs of each ruleset invocation. Any number of rulesets can be profiled at once. The collected statistics include the total time spent within a call and the number of calls made. A sample of the results is shown below using the "players" program.

<21> profile "players".

Breaking 'FILE: "players"'...
 TO FIND BASKETBALL PLAYERS -- broken.
 TO DECIDE IF A PERSON IS TALL -- broken.

TO GENERATE A MAN -- broken.
 TO GENERATE THE HEIGHT OF A PERSON -- broken.
 Done breaking.

<22> find basketball players.
 JACK is a basketball player.
 <23> profile report.

Timing results for:

-
-
- [1] TO FIND BASKETBALL PLAYERS
 - [2] TO DECIDE IF A PERSON IS TALL
 - [3] TO GENERATE A MAN
 - [4] TO GENERATE THE HEIGHT OF A PERSON

	Ruleset	Total Time (sec)	Number of Calls	Time per Call (sec)	Percent of Total Time
[1]	FIND	0.017	1	0.017	5
[2]	TALL	0.102	3	0.034	33
[3]	MAN	0.102	1	0.101	33
[4]	HEIGHT	0.085	3	0.028	27
<hr/>					
Totals:		0.306	8	0.038	

When nesting of profiled rulesets occurs, time is charged only to the innermost invocation. Further, the timing statistics are cumulative until reset by **profile reset**.

14.5 RESTORING BROKEN RULESETS

The **unbreak** and **untrace** procedures restore broken rulesets to their original state. If a ruleset was edited while broken, the edits are not lost when the ruleset is unbroken.

Unbreak and **untrace** actually do exactly the same thing; both procedures are provided for historical reasons. These procedures are used to restore any ruleset broken by either the break, trace, or profile facilities.

14.6 BREAK PACKAGE OPERATIONS

In the following operations, a *filespec* is a reference to a filesegment using either the formal or shorthand notation.

NOTE: To break, trace, or profile a ruleset, the ruleset need not be noticed nor even exist. If undefined, a dummy definition is supplied. To break, trace, or profile individual rules within a ruleset, the ruleset *must* be noticed.

break [*a filespec*]

If given no argument, breaks all noticed program files. If applied to a program file, redefines all rulesets of that file so that their invocation causes control to enter a break. If applied to a ruleset, redefines just that ruleset to throw control into a break loop when invoked.

If applied to a generator ruleset, also breaks the produced demon of the same name. This allows the user to examine each element before it is actually produced.

If applied to a rule or sequence of rules within a ruleset, redefines the ruleset such that a message is printed when the ruleset is entered and control is passed to a break loop before each broken rule is executed.

trace [*a filespec*]

Similar to **break**, with the exception that, rather than entering a break loop, rulesets within *filespec* are redefined to print messages prior to and following invocation. Trace messages designate the calling level, the calling form, and the value being returned.

profile [*a filespec*]

Like **break** and **trace**, this procedure redefines the rulesets within *filespec*. Profiling keeps track of the frequency and execution time of selected filesegments. Performance information is collected incrementally. Re-profiling a ruleset resets the profile information associated with it.

profile report

Displays the profiling information collected so far.

profile reset

Reinitializes all profiling information.

unbreak [*a filespec*]**untrace** [*a filespec*]

Restores all broken rulesets (i.e., redefined by **break**, **trace**, or **profile**) within *filespec*. If no *filespec* is given, unbreaks all broken rulesets. Unbreaking a profiled ruleset causes all

profiling information on that ruleset to vanish.

NOTE: It is not possible to unbreak a single rule in a ruleset without unbreaking the entire ruleset.

APPENDIX A: EXAMPLE PROGRAMS

Examples provide a good introduction to ROSIE. The following annotated ROSIE systems serve to illustrate several of the concepts discussed in the manual. Because ROSIE is unlike any other high-level language, it requires some practice before the user begins to approach problems in a "ROSIE-compatible" manner. Examining example systems is the first step in learning how to program in ROSIE in an intelligible and effective manner.

The first example system, FORTUNE, demonstrates a simple application of ROSIE that exercises many of its basic data manipulation capabilities. The second example, POIROT, demonstrates the use of alternate databases as a method for describing distinct belief spaces. The third example, ANIMAL, shows how to build a control structure on top of ROSIE. The first two examples were adapted for ROSIE 3.0 from an earlier ROSIE document entitled "Programming in ROSIE: An Introduction by Means of Examples," (Fain et al., 1982); readers interested in seeing other (though outdated) examples of ROSIE programming techniques are referred to this document.

NOTE: These and other ROSIE demo programs are provided with the standard ROSIE distribution and should be on-line at your installation. See your site consultant for information on how they can be accessed.

FORTUNE -- THE BASICS

The FORTUNE demo system performs a mock portfolio analysis. FORTUNE was written several years ago to demonstrate ROSIE's capabilities to the editors of Fortune magazine.

Developing a knowledge-based system for any task requires designating the pertinent information needed and determining how it should be used. In ROSIE, this often means deciding what should go into the database and what should be encapsulated, procedurally, into rulesets.

For the task of FORTUNE, the problem statement was given as shown below:

Renewable energy, electronics, and optics are high-technology areas of investment, and genetic engineering is a high-technology area of investment that is speculative. Communications, photography, automotive, and machine tools are conservative areas of investment, and petroleum is an area of investment that affords rapid write-off but is a little risky. Petroleum and renewable energy are current topics of legislation.

John Doe owns an investment portfolio that includes renewable energy, petroleum, automotive, and communications, and Mary Jones owns an investment portfolio that includes renewable energy, genetic engineering, and optics. John Doe and Mary Jones are investors. John Doe's line of credit is \$20,000 and Mary Jones' is \$10,000.

Any area of investment is a stock and any stock that is a current topic of legislation is volatile. Investors who hold high-technology stock are generally interested in productivity, and investors who hold speculative stock will be interested in robotics and artificial intelligence.

ROSIE is an innovation in both productivity and artificial intelligence, which are technology areas.

For each investor, we want to be able to list the holdings for that investor, giving such related information as the investor's name, and a title such as "Current holdings" and marking as "VOLATILE" all volatile stocks held by the investor.

We want to do a profile of Mary Jones. This means listing any speculative stocks she holds with an appropriate title. If she holds no speculative stocks, an appropriate message should be given.

We should send a separate technology-area bulletin to every investor interested in that area announcing each technology

innovation in that area. One bulletin per innovation, please. If the same innovation relates to multiple areas, send multiple bulletins with appropriately differing titles.

Finally, we want to try to find a customer for ROSIE. This investor must be interested in productivity and artificial intelligence. He or she must have a line of credit exceeding \$1,000. If a customer is found, record the reasons for selecting him or her as a ROSIE prospect. Display the customer and the reasons.

The problem definition clearly separates the static from the procedural. We encode the static information by adding facts to the database. In the rules seen below, note the close correspondence between the text and its codification in ROSIE.

[FORTUNE DATA]

[rule 1] Assert any high-technology area of investment is an area of investment that does involve high technology.

[rule 2] Assert any speculative area of investment is an area of investment that is speculative.

[rule 3] Assert any conservative area of investment is an area of investment that is conservative.

[rule 4] Assert each of renewable energy, electronics and optics is a high-technology area of investment and genetic engineering is a high-technology area of investment that is speculative.

[rule 5] Assert each of communications, photography, automotive and machine tools is a conservative area of investment and petroleum is an area of investment that does afford rapid write-off and that is a little risky.

[rule 6] Assert each of petroleum and renewable energy is a current topic of legislation.

[rule 7] Assert each of John Doe and Mary Jones is an investor.

[rule 8] Assert John Doe does own an investment portfolio that does include each of renewable energy, petroleum, automotive and communications and Mary Jones does own an investment portfolio that does include each of renewable energy, genetic engineering and optics.

[rule 9] Let John Doe's line-of-credit be \$ 20000 and

Mary Jones's line-of-credit be \$ 10000.

- [rule 10] Assert any area of investment is a stock and any stock that is a current topic of legislation is volatile.
- [rule 11] Assert any stock that does involve high technology is a high-technology stock.
- [rule 12] Assert any stock that is speculative is a speculative stock.
- [rule 13] Assert any stock that is conservative is a conservative stock.
- [rule 14] Assert every investor who is a holder of some high-technology stock will be interested in productivity and every investor who is a holder of some speculative stock will be interested in each of robotics and artificial intelligence.
- [rule 15] Assert ROSIE is an innovation in each of productivity and artificial intelligence and each of productivity and artificial intelligence is a technology area.

Rules 1, 2, and 3 make use of class elements (introduced by **any**) to define general attributes of the three areas of investment mentioned in the problem statement. Note that in earlier releases of ROSIE, similar attributes would have been generated automatically, e.g., anything that was a **speculative area of investment** would also be **speculative**. Unfortunately, that feature also made deductions that were not exactly accurate, such as anything that is a **high-technology area of investment** is also **high-technology** (used as an adjective attribute). Thus, in ROSIE 3.0 the attributes of a class must be stated explicitly as they are here.

Rules 4, 5, and 6 encapsulate the first paragraph of the problem statement. Notice the use of the **each of** iterator to establish identical *is-a* relations for a number of elements. While each of these assertions could have been made separately with the same end result, the use of **each of** makes them more compact and easy to read, i.e., more like English and less like a programming language. Note also that in the rules above, indenting helps to clarify the extent of each assertion.

Rule 7 establishes the relation **is an investor** for John Doe and Mary Jones. This will implicitly provide a link between those rules that apply to investors in general and the information stored for each of the individuals.

Rules 8 and 9 correspond to the second paragraph of the problem statement. The **let** action in rule 9 establishes \$ 20000 uniquely as the **line-of-credit of John Doe** and \$ 10000 as the **line-of-credit of Mary Jones**. Note that **<term>'s <description>** expands into **<description> of**

<term>, attaching an additional prepositional phrase to *description*. Note also the necessity to hyphenate *line-of-credit*. If there were no hyphenation, e.g., *John Doe's line of credit*, then *line* would have two prepositional phrases attached, i.e., *of credit* and *of John Doe*; since both use the same preposition, the parser would call a syntax error. Hyphens make *line-of-credit* all one word, circumventing the parsers recognition of the preposition *of*.

Rules 10 through 13 again use class elements to make a link between the class **stock** and the class **area of investment**. ROSIE uses such links for deductive information retrieval. For instance, to get a **high-technology stock**, ROSIE sees that this could be **any stock which does involve high technology**. Then ROSIE finds that **any area of investment** is a stock, **any high-technology area of investment** is an area of investment, and **renewable energy** is a high-technology area of investment. Since **any high-technology area of investment** does involve high technology, **renewable energy** can be deduced as a high-technology stock.

Rule 14 could likewise have been expressed using class elements. Instead, a quantified descriptive term (introduced by **every**) was used, resulting in instance-specific rather than class-specific relations to appear in the database. The use of **every** tells us that the programmer does not anticipate any more investors being added to the database. Since **every** creates an explicit rather than implicit link to the members of a class at the time of evaluation, it would be more appropriate to use class elements when all members of a class are not known a priori.

Finally, rule 15 turns that fourth paragraph of the problem statement into ROSIE-compatible form.

The database resulting from the execution of rules 1 through 15 would have the following appearance:

[GLOBAL Database]

ANY STOCK WHICH IS A CURRENT TOPIC OF LEGISLATION IS VOLATILE.
 ANY CONSERVATIVE AREA OF INVESTMENT IS CONSERVATIVE.
 GENETIC ENGINEERING IS SPECULATIVE.
 ANY SPECULATIVE AREA OF INVESTMENT IS SPECULATIVE.
 MARY JONES WILL BE INTERESTED IN ARTIFICIAL INTELLIGENCE.
 MARY JONES WILL BE INTERESTED IN ROBOTICS.
 JOHN DOE WILL BE INTERESTED IN PRODUCTIVITY.
 MARY JONES WILL BE INTERESTED IN PRODUCTIVITY.
 MARY JONES DOES OWN PORTFOLIO #2.
 JOHN DOE DOES OWN PORTFOLIO #1.
 PORTFOLIO #2 DOES INCLUDE OPTICS.
 PORTFOLIO #2 DOES INCLUDE GENETIC ENGINEERING.
 PORTFOLIO #2 DOES INCLUDE RENEWABLE ENERGY.
 PORTFOLIO #1 DOES INCLUDE COMMUNICATIONS.
 PORTFOLIO #1 DOES INCLUDE AUTOMOTIVE.
 PORTFOLIO #1 DOES INCLUDE PETROLEUM.

PORTFOLIO #1 DOES INCLUDE RENEWABLE ENERGY.
 PETROLEUM DOES AFFORD RAPID WRITE-OFF.
 ANY HIGH-TECHNOLOGY AREA OF INVESTMENT DOES INVOLVE HIGH TECHNOLOGY.
 ARTIFICIAL INTELLIGENCE IS A TECHNOLOGY AREA.
 PRODUCTIVITY IS A TECHNOLOGY AREA.
 ROSIE IS AN INNOVATION IN ARTIFICIAL INTELLIGENCE.
 ROSIE IS AN INNOVATION IN PRODUCTIVITY.
 ANY STOCK WHICH IS CONSERVATIVE IS A CONSERVATIVE STOCK.
 ANY STOCK WHICH IS SPECULATIVE IS A SPECULATIVE STOCK.
 ANY STOCK WHICH DOES INVOLVE HIGH TECHNOLOGY IS A
 HIGH-TECHNOLOGY STOCK.
 ANY AREA OF INVESTMENT IS A STOCK.
 \$ 15000 IS A LINE-OF-CREDIT OF MARY JONES.
 \$ 20000 IS A LINE-OF-CREDIT OF JOHN DOE.
 PORTFOLIO #2 IS AN INVESTMENT PORTFOLIO.
 PORTFOLIO #1 IS AN INVESTMENT PORTFOLIO.
 MARY JONES IS AN INVESTOR.
 JOHN DOE IS AN INVESTOR.
 RENEWABLE ENERGY IS A CURRENT TOPIC OF LEGISLATION.
 PETROLEUM IS A CURRENT TOPIC OF LEGISLATION.
 PETROLEUM IS A LITTLE RISKY.
 MACHINE TOOLS IS A CONSERVATIVE AREA OF INVESTMENT.
 AUTOMOTIVE IS A CONSERVATIVE AREA OF INVESTMENT.
 PHOTOGRAPHY IS A CONSERVATIVE AREA OF INVESTMENT.
 COMMUNICATIONS IS A CONSERVATIVE AREA OF INVESTMENT.
 GENETIC ENGINEERING IS A HIGH-TECHNOLOGY AREA OF INVESTMENT.
 OPTICS IS A HIGH-TECHNOLOGY AREA OF INVESTMENT.
 ELECTRONICS IS A HIGH-TECHNOLOGY AREA OF INVESTMENT.
 RENEWABLE ENERGY IS A HIGH-TECHNOLOGY AREA OF INVESTMENT.
 PETROLEUM IS AN AREA OF INVESTMENT.
 ANY CONSERVATIVE AREA OF INVESTMENT IS AN AREA OF INVESTMENT.
 ANY SPECULATIVE AREA OF INVESTMENT IS AN AREA OF INVESTMENT.
 ANY HIGH-TECHNOLOGY AREA OF INVESTMENT IS AN AREA OF INVESTMENT.

Note that the appearance of

MARY JONES WILL BE INTERESTED IN PRODUCTIVITY

came about through the use of **every** in rule 14.

The remainder of problem statement requires a variety of tasks to be performed. These tasks are codified as the ROSIE rulesets seen below.

[FORTUNE RULESETS]

To decide if a given investor is a holder of a given stock:

- [1] If the given investor does own any investment portfolio that
 does include the given stock,

conclude true, otherwise conclude false.

End.

To decide if a given stock is held by a given investor:

- [1] If the given investor is a holder of the given stock,
conclude true, otherwise conclude false.

End.

[----- Show investors' holdings -----]

To list-holdings:

- [1] For each investor,
send "{cr}{that investor}'s current holdings:{cr}" and
for each stock that is held by that investor,
send " {that stock}" and
if that stock is volatile then send " -- VOLATILE!!" and
send "{cr}".

End.

[----- Show an investor's speculative stocks -----]

To list-speculatives of a given investor:

- [1] If the given investor is a holder of any speculative stock, then
send "{cr}{that investor}'s speculative holdings:{cr}" and
send " {every speculative stock that is held by that investor}{cr}",
otherwise
send "{cr}{the given investor} has no speculative holdings.{cr}".

End.

[----- Notify investors of innovations in their interest areas -----]

To announce innovations:

- [1] Send "{cr}".

- [2] For each technology area,
send "**** {that area} BULLETIN to: {every investor who will
be interested in that area} -- Find out about {every
innovation in that area}{cr}".

End.

[---- Look for someone with good credit who is interested in ROSIE ----]

To locate customers:

[1] If there is an investor who will be interested in each of productivity and artificial intelligence and who has a line-of-credit that is greater than \$ 1000,
 let that investor be ROSIE's first customer and
 assert each of 'that investor does have good credit' and
 'that investor does like each of artificial
 intelligence and productivity'
 is a reason for 'that investor is a customer of ROSIE'.

[2] Send "{cr}ROSIE's first customer is {ROSIE's first customer}"

 The reasons for this include:{cr}" and

 send " {every reason for 'that customer is a customer of ROSIE'}{cr}".

[4] Send "{cr}".

End.

[----- Procedure to demo the system -----]

To demonstrate:

[1] Send "{cr}[rule 1] List-holdings.{cr}" and list-holdings.

[2] Send "{cr}[rule 2] List-speculatives of Mary Jones.{cr}" and
 list-speculatives of Mary Jones.

[3] Send "{cr}[rule 3] Announce innovations.{cr}" and announce innovations.

[4] Send "{cr}[rule 4] Locate a customer.{cr}" and locate a customer.

End.

The FORTUNE program consists of three pieces: four procedural rulesets (one for each of the four tasks outlined in the problem statement), two support predicate rulesets, and a "driver" procedural ruleset that organizes the actual demonstration.

The predicate ruleset,

 To decide if a given investor is a holder of a given stock:

shows how a predicate can be used when the criteria for class membership cannot be encapsulated in a proposition and asserted in the database. For the **is a holder of** relation to be true, two dependent database relations must coexist: (1) the investor must own an investment portfolio, and (2) that portfolio must include the desired stock.

The predicate ruleset,

To decide if a given stock is held by a given investor:

essentially coerces an equivalence between the **is a holder of** and the **is held by** relations. The manner of employing predicates allows one to express the same idea naturally in different contexts; predicates can provide equivalent semantics for different ROSIE relations.

The procedural rulesets,

To list-holdings:

To list-speculatives of a given investor:

To announce innovations:

illustrate the primary mechanism for output in ROSIE--the **send** procedure and patterns for text formatting. In particular, these rulesets illustrate the new extended string syntax being introduced in ROSIE 3.0. The first and third ruleset use the **for each** action to generate the elements of the set conforming to the given description, iteratively printing a message for each such element.

The procedural ruleset,

To locate customers:

exemplifies a number of interesting points. First it shows the use of **each of** in testing, e.g., the same investor who is interested in productivity must also be interested in artificial intelligence. Note also that this ruleset uses **\$ 1000** in a comparison. This is called a labeled constant and can be added, subtracted, compared, etc., in the same manner as 1000 alone. Finally, this ruleset also incorporates the use of intentional propositions (proposition between single quotes). Intentional propositions let you manipulate declarative knowledge as data.

The procedural ruleset,

To demonstrate:

organizes the demonstration itself. Once at the top level of ROSIE, we can load the FORTUNE program and type **demonstrate**, with the following results:

<4> demonstrate.

[rule 1] List-holdings.

MARY JONES's current holdings:
GENETIC ENGINEERING
OPTICS
RENEWABLE ENERGY -- VOLATILE!!

JOHN DOE's current holdings:
PETROLEUM -- VOLATILE!!
AUTOMOTIVE
COMMUNICATIONS
RENEWABLE ENERGY -- VOLATILE!!

[rule 2] List-speculatives of Mary Jones.

MARY JONES's speculative holdings:
GENETIC ENGINEERING

[rule 3] Announce innovations.

*** ARTIFICIAL INTELLIGENCE BULLETIN to: MARY JONES -- Find out
about ROSIE
*** PRODUCTIVITY BULLETIN to: MARY JONES -- Find out about ROSIE
*** PRODUCTIVITY BULLETIN to: JOHN DOE -- Find out about ROSIE

[rule 4] Locate a customer.

ROSIE's first customer is MARY JONES
The reasons for this include:
'MARY JONES DOES LIKE PRODUCTIVITY'
'MARY JONES DOES HAVE GOOD CREDIT'
'MARY JONES DOES LIKE AI'

POIROT -- ALTERNATE DATABASES

The next demo system, POIROT, introduces ROSIE to the world of the detective. The motivation behind this system, however, is not to explore the principles of investigation, but to examine the use of alternate databases within a multiple database structure.

The problem statement in this domain is straightforward:

Given some set of facts and some set of participants, the detective, POIROT, must uncover the information necessary to deduce which of the participants might be guilty.

POIROT uncovers information by mediating a dialogue between the user and each participant. POIROT then uses the information gleaned from the interrogation to make his deductions, i.e., the user asks the questions and POIROT makes the inferences.

In any scenario there will be only one victim, who is found either dead, shot, or stabbed. Each potential suspect has his or her own viewpoint of and knowledge about the situation.

Note that in terms of implementation, the last sentence requires a method for simulating the privacy of each participant's memory as well as some mechanism for simulating the question/answer protocol of interrogation.

The mechanism for simulating "belief" in this system is quite simple. Segregation of idiosyncratic knowledge is accomplished with alternate databases. The knowledge that is unique to each participant is stored in a distinct alternate database for that participant. Information that is available to everyone is stored in the global database. At the beginning of the case, we set up the following scenario from rules in four different program files:

[SCENARIO]

[rule 1] Assert both any woman and any man is a person.

[rule 2] Assert each of John and Poirot is a man.

[rule 3] Assert each of Mary and Sara is a woman.

[rule 4] Assert Mary is rich.

[rule 5] Assert Mary is found dead.

[rule 6] Let the detective be Poirot.

[JOHN]

[rule 1] Activate John's world.
[rule 2] Assert John does need money.
[rule 3] Assert John is married to Mary.
[rule 4] Assert John did love Mary.
[rule 5] Deactivate.

[SARA]

[rule 1] Activate Sara's world.
[rule 2] Assert Sara is a sister of Mary.
[rule 3] Assert Sara does love John.
[rule 4] Assert John did not love Mary.
[rule 5] Assert John does love Sara.
[rule 6] Deactivate.

[POIROT]

[rule 1] Activate Poirot's world.
[rule 2] Assert Sara is involved and John is involved.
[rule 3] Deactivate.

The first file establishes general definitions and facts about the case in the global database. Each of the other files begins with an action to activate an alternate database, which acts as the world of a particular participant. All subsequent assertions affirm relations exclusively in that database. The generator '**to generate the world of an individual**' (defined below) causes **John's world** to evaluate to **JOHNS**, and so on--thus, the name of the first alternate database is actually **JOHNS**. Note that the program file defining this generator must be loaded before the scenario files.

The presence of ROSIE's multiple database structure enables the representation of contradictions, i.e., as long as they reside in different databases. For instance, **Sara** believes that '**John does not love Mary**' (rule 4 of SARA), while **John** believes the opposite (rule 4 of JOHN). Because ROSIE attempts to maintain consistency within a database, both beliefs could not be in one database unless they were treated as propositions in the following format

Sara does believe 'John does not love Mary'
John does believe 'John does love Mary'

Thus, the database for an individual acts as the belief space for that individual it represents.

The code for POIROT is given below. Note that rulesets do not reside in any database. They are accessible regardless of which database is active.

[DETECTIVE RULESETS]

[----- General relation about participants -----]

To decide if a character is a victim:

- [1] If the character is found either dead, stabbed, or shot,
conclude true, otherwise conclude false.

End.

To generate the victim:

- [1] If any thing is found either dead, stabbed, or shot,
produce that thing.

End.

To decide if a character is related to an individual:

- [1] If the character is married to the individual or
the individual is married to that character,
note 'the character is married to the individual' and
conclude true.
- [2] If the character is a sister of the individual,
note 'the character is a sister of the individual' and
conclude true.
- [3] If the character is a brother of the individual,
note 'the character is a brother of the individual' and
conclude true.
- [4] If the character is a mother of the individual,
note 'the character is a mother to the individual' and
conclude true.
- [5] If the character is a father of the individual,

note 'the character is a father to the individual' and
conclude true.

[6] If the character is a son of the individual,
note 'the character is a son of the individual' and
conclude true.

[7] If the character is a daughter of the individual,
note 'the character is a daughter of the individual' and
conclude true.

End.

[----- Procedure to demo the system -----]

To detect:

[1] Swap in Poirot's world.

[2] For each person who is involved,
send "{cr}Interrogating {that person}{cr}" and
interrogate that person.

[3] Send "{cr}Suspects: Motives--{cr}".

[4] For each suspect,
send "{cr} {that suspect}: {every reason for suspicion
of that suspect}".

[5] Send "{2 crs}".

End.

[----- Solicit questions from the user -----]

To interrogate a character:

Private: a reply.

Execute cyclically.

[1] Send "{cr}> ".

[2] Read "{anything (bind the reply)){cr}".

[3] Match the lowercase of the reply:

{""}

```

    return;

    {"interrogee?"}

    display the character;

    {"is it that ", anything (bind PROP), "?"}

    question the character about PROP;

    default: send "MUST USE THE FORM: Is it that <proposition>?{cr}".

End.

```

To question a character about a query:

- [1] Swap in the character's world.
- [2] If there is a proposition from "{the query}" and
that proposition is provably true,
note that proposition and repeat facts,
otherwise display UNKNOWN.

End.

[----- Deduce who could be the murderer -----]

To generate a suspect:

- [1] For each person (P) who is involved,
 - if P does love any person who is married to the victim,
assert jealousy is a reason for suspicion of P, and
 - if P does need money and the victim is rich and
P is related to that victim,
assert monetary gain is a reason for suspicion of P, and
 - if P was rejected by either the victim or any person
who does love that victim,
assert revenge is a reason for suspicion of P, and
 - if there is a reason for suspicion of P,
produce P.

End.

[----- Remember newly acquired information -----]

To note a statement:

[1] Add the statement to facts.

End.

[----- Relate what Poirot now knows -----]

To repeat facts:

[1] For each affirmed proposition from facts,
 add that proposition to Poirot's world and
 echo that proposition.

[2] Clear facts.

End.

To echo a fact:

[1] Send "Yes, {the fact}.\n".

End.

[----- Returns a database name for a character -----]

To generate a world of a character:

[1] Produce the name from "{the character}s".

End.

As with the FORTUNE example, it is convenient to think of this program as consisting of three parts: First, some support predicates and generators; second, the driver routine; and third, the rulesets invoked by the driver.

Among the support routines is the definition of what it means to be a **victim**. Note that this definition could easily have been defined by asserting a proposition of the form

any person who is found either dead, shot,
 or stabbed is a victim

This approach was *not* taken for reasons that will become clear. Given this latter approach, in order to find **the victim**, it is necessary to iterate through each instance of the class **person** until an instance is found that is either dead, shot, or stabbed. The approach seen in

POIROT takes advantage of the fact that there will be only one instance of the relations

element is found dead
element is found shot
element is found stabbed

affirmed at any given time, and that *element* will be the victim. Thus, in the predicate ruleset,

To decide if a character is a victim:

it is only necessary to check if the character is found dead, shot, or stabbed. The generator ruleset,

To generate the victim:

operates in a similar fashion, but uses a trick to retrieve *element*. Testing the proposition in rule [1] first finds the instance of the **found** relation that is affirmed and then tests if *element* is a thing. The trick is that all elements satisfy this predicate, allowing the ruleset to produce the victim after accessing the database a maximum of four times. The bottom line of this discussion is the following, since there is only one victim and potentially many persons, the approach taken in POIROT is the most efficient.

The procedural ruleset,

To detect:

acts as the driver routine. Rule [1] activates the database for the detective; this is done with **swap in** so that the originally active database (in this case the global database) will be reactivated automatically when the ruleset terminates. Rule [2] applies the **interrogate** procedure to each person the detective thinks is involved. After interrogation, rule [4] lists the possible suspects and their motives.

The **interrogate** procedure runs interactively, allowing the user to ask the questions. This ruleset is running under a cyclic execution monitor, which means its rules are executed from top-to-bottom over and over until execution of the **return** procedure. This ruleset also uses the private class **reply**, which will be bound to the substring matched by the pattern in rule [2]. This ruleset works by prompting the user for a question in rule [1] and then reading one line of input from the user in rule [2]. The user's options are a carriage return, indicating the interrogation is over; the question "Interrogee?", which names the person currently under questioning; or a question of the form

Is it that *proposition*?

Note that the input is first converted to lower case to standardize the form of comparison. The substring matching *proposition* is then sent to the **question** procedure, which queries the interrogee.

The **question** procedure first temporarily activates the database of the interrogee. In rule [2], the question is evaluated. First, rule [2] tries to turn the string representing the question into an intentional proposition, if possible, and then it tests the truth or falsity of this proposition given the current state of the database. Given that it tests true, the ruleset notes and lists that fact as well as any other facts that were discovered as a result. Facts are incrementally stored in the **facts** database by the **note** procedure and listed by the **repeat** procedure, which also moves those facts to the detective's database.

The detective goes to work in the generator ruleset,

To generate a suspect:

which is called after everyone involved in the case has been interrogated. For each person involved, this ruleset amasses motives based on its "knowledge" of the psychology of crime. In this case, three of the more common motives for murder (i.e., jealousy, monetary gain, and revenge) are represented.

The following interactions show POIROT at work on the case described in the scenario given above:

<7> ?

[GLOBAL Database]

MARY IS FOUND DEAD.

MARY IS RICH.

POIROT IS A DETECTIVE.

SARA IS A WOMAN.

MARY IS A WOMAN.

POIROT IS A MAN.

JOHN IS A MAN.

ANY MAN IS A PERSON.

ANY WOMAN IS A PERSON.

<8> Johns?

[JOHNS Database]

JOHN IS MARRIED TO MARY.

JOHN DOES NEED MONEY.

JOHN DID LOVE MARY.

<9> Saras?

[SARAS Database]

JOHN DOES LOVE SARA.

SARA DOES LOVE JOHN.

JOHN DID NOT LOVE MARY.

SARA IS A SISTER OF MARY.

<10> Poirots?

[POIROTS Database]
JOHN IS INVOLVED.
SARA IS INVOLVED.

<11> Detect.

Interrogating JOHN

> Is it that John is related to the victim?
Yes, 'JOHN IS RELATED TO MARY'.
Yes, 'JOHN IS MARRIED TO MARY'.

> Is it that John did love Mary?
Yes, 'JOHN DID LOVE MARY'.

> Is it that John does need money?
Yes, 'JOHN DOES NEED MONEY'.

> Is it that John does love Sara?

UNKNOWN

>

Interrogating SARA

> Is it that Sara is related to the victim?
Yes, 'SARA IS RELATED TO MARY'.
Yes, 'SARA IS A SISTER OF MARY'.

> Is it that Sara does love John?
Yes, 'SARA DOES LOVE JOHN'.

> Is it that John does love Sara?
Yes, 'JOHN DOES LOVE SARA'.

>

Suspects: Motives--

JOHN: MONETARY GAIN
SARA: JEALOUSY

<12> Poirots?

[POIROTS Database]
JOHN DOES LOVE SARA.
SARA DOES LOVE JOHN.
JOHN DOES NEED MONEY.

JOHN DID LOVE MARY.
JEALOUSY IS A REASON FOR SUSPICION OF SARA.
MONETARY GAIN IS A REASON FOR SUSPICION OF JOHN.
SARA IS A SISTER OF MARY.
JOHN IS MARRIED TO MARY.
SARA IS RELATED TO MARY.
JOHN IS RELATED TO MARY.
JOHN IS INVOLVED.
SARA IS INVOLVED.

As a final note, even though the question, "Is it that John does love Sara?" elicits a response of UNKNOWN from the interrogation of John's database and "Yes, 'JOHN DOES LOVE SARA'." from Sara's database, the statement appears as a fact in Poirot's database. This illustrates that the mechanism for segregating beliefs used in POIROT is too simple for even this small exercise. One next step that might be taken to relieve the problem seen here is to develop a mechanism for describing certainty as well as belief, however, that is outside the scope of this example.

ANIMAL -- EMBEDDED CONTROL STRUCTURES

The last demo system, ANIMAL, illustrates how to build a simple control structure on top of ROSIE. In this case, the control structure is a production system monitor that applies rules in a goal-directed (i.e., backward-chaining) manner.

ANIMAL is derived from a system described in (Winston, 1979) in which "Robbie the robot" develops a set of rules for classifying animals in the zoo. Applying these rules in a backward-chaining manner allows Robbie to determine which animal he is seeing. Thus, the problem statement for ANIMAL is

Given Robbie's rules, figure out which of the possible seven animals the user is thinking of.

For the purposes of ANIMAL, each rule will be represented as a set of assertions about its *preconditions* and *conclusions*. For instance, the rule

```
IF    the animal is a mammal
      and it eats meat,
THEN  it is a carnivore.
```

will be represented by the three propositions

```
'animal is a mammal' is a precondition of rule #6
'animal does eat meat' is a precondition of rule #6
'animal is a carnivore' is a conclusion of rule #6
```

These propositions are asserted by a file rule of the form

```
[rule 6] Let the conclusion of a new rule be 'animal is a carnivore' and
assert each of 'animal does eat meat' and
               'animal is a mammal'
is a precondition of that rule.
```

Note that a **new rule** creates the name of the rule (i.e., the comment **[rule 6]** has no bearing). This is **rule #6** because the preceding five file rules created five new ANIMAL rules.

There are 15 ANIMAL rules in all, defined as follows:

```
[ ANIMAL DATA ]
```

```
[rule 1] Let the conclusion of a new rule be 'animal is a mammal' and
assert 'animal does give milk'
is a precondition of that rule.
```

```
[rule 2] Let the conclusion of a new rule be 'animal is a mammal' and
```

assert 'animal does have hair'
is a precondition of that rule.

[rule 3] Let the conclusion of a new rule be 'animal is a bird' and
assert 'animal does lay eggs'
is a precondition of that rule.

[rule 4] Let the conclusion of a new rule be 'animal is a bird' and
assert 'animal does have feathers'
is a precondition of that rule.

[rule 5] Let the conclusion of a new rule be 'animal is a carnivore' and
assert each of 'animal does have pointed teeth',
 'animal does have claws',
 'animal does have its eyes pointed ahead' and
 'animal is a mammal'
is a precondition of that rule.

[rule 6] Let the conclusion of a new rule be 'animal is a carnivore' and
assert each of 'animal does eat meat' and
 'animal is a mammal'
is a precondition of that rule.

[rule 7] Let the conclusion of a new rule be 'animal is an ungulate' and
assert each of 'animal does chew cud' and
 'animal is a mammal'
is a precondition of that rule.

[rule 8] Let the conclusion of a new rule be 'animal is an ungulate' and
assert each of 'animal does have hoofs' and
 'animal is a mammal'
is a precondition of that rule.

[rule 9] Let the conclusion of a new rule be 'animal is a cheetah' and
assert each of 'animal does have dark spots',
 'animal is tawny colored' and
 'animal is a carnivore'
is a precondition of that rule.

[rule 10] Let the conclusion of a new rule be 'animal is a tiger' and
assert each of 'animal does have black stripes',
 'animal is tawny colored' and
 'animal is a carnivore'

is a precondition of that rule.

[rule 11] Let the conclusion of a new rule be 'animal is a giraffe' and
assert each of 'animal does have dark spots',
 'animal is tawny colored',
 'animal does have long legs-and-neck', and
 'animal is an ungulate'
is a precondition of that rule.

[rule 12] Let the conclusion of a new rule be 'animal is a zebra' and
assert each of 'animal is white with black stripes', and
 'animal is an ungulate'
is a precondition of that rule.

[rule 13] Let the conclusion of a new rule be 'animal is an ostrich' and
assert each of 'animal is black-and-white',
 'animal does have long legs-and-neck',
 'animal does not fly' and
 'animal is a bird'
is a precondition of that rule.

[rule 14] Let the conclusion of a new rule be 'animal is a penguin' and
assert each of 'animal is black-and-white',
 'animal does swim',
 'animal does not fly' and
 'animal is a bird'
is a precondition of that rule.

[rule 15] Let the conclusion of a new rule be 'animal is an albatross' and
assert each of 'animal is a good flyer' and
 'animal is a bird'
is a precondition of that rule.

[rule 16] Assert each of 'animal is a cheetah',
 'animal is a tiger',
 'animal is a giraffe',
 'animal is a zebra',
 'animal is an ostrich',
 'animal is a penguin' and
 'animal is an albatross'
is a hypothesis.

The last rule defines the seven possible hypotheses about which the
system can make any conclusions.

Given these rules, ANIMAL applies them using a very simple backward-chaining strategy. To begin, it iterates through the hypotheses until it finds one that it can prove. It then prints that that hypothesis is true and quits. To prove a hypothesis, ANIMAL looks for a rule that concludes that hypothesis and then treats the preconditions of that rule as new hypotheses to prove--if the preconditions of a rule are true, then its conclusion is true. If a hypothesis is not a conclusion of a rule, then ANIMAL asks the user a yes/no question to confirm or deny the hypothesis. Thus, ANIMAL starts a game of "20 questions" that leads to identifying the animal.

The rulesets used for applying the ANIMAL rules in this manner are relatively simple and straightforward. They are defined below.

[ANIMAL RULESETS]

[----- Procedure to demo the system -----]

To diagnose:

- [1] For each hypothesis until that hypothesis is provably true,
 apply the rule of which that hypothesis is a conclusion.
- [2] If there is a hypothesis that is provably true,
 send "{cr}Hypothesis: {that hypothesis}{cr}{cr}",
 otherwise,
 send "{cr}No hypothesis can be confirmed.{cr}{cr}".

End.

[----- Procedure that tests a rule -----]

To apply a rule:

- [1] Choose situation:
 - if every precondition of the rule is true,
 assert the conclusion of the rule is provably true;
 - if there is no precondition (of the rule) that is unknown,
 assert the conclusion of the rule is provably false.

End.

[----- Figure out how to test a hypothesis -----]

To decide if a hypothesis is true:

- [1] If the hypothesis is unknown, return.
- [2] If the hypothesis is provably true, conclude true.
- [3] If the hypothesis is provably false, conclude false.
- [4] Choose situation:

if the hypothesis is a conclusion of any rule,
 then for each rule of which that hypothesis is a conclusion
 until that hypothesis is provably either true or false,
 apply that rule, and
 if the hypothesis is provably true,
 conclude true;

if the hypothesis is confirmed, conclude true;

default: unless the hypothesis is [now] unknown, conclude false.

End.

[----- Query the user for yes/no answer -----]

To decide if a hypothesis is confirmed:

Private: a reply.

Execute cyclically.

[1] Send "{cr}Q: {the query for the hypothesis}".

[2] Send "{cr}A: ".

[3] Read "{anything (bind the reply)){cr}".

[4] Select the uppercase of the reply:

<"YES"> If the hypothesis is not negated,
 assert the hypothesis is provably true and conclude true,
 otherwise,
 assert the hypothesis is provably false and conclude false;

<"NO"> If the hypothesis is not negated,
 assert the hypothesis is provably false and conclude false,
 otherwise,
 assert the hypothesis is provably true and conclude true;

<"?"> Assert the hypothesis is unknown and return;

default: send "{cr}Type YES or NO (or ? for unknown){cr}".

End.

ANIMAL consists of four rulesets: one procedural ruleset that acts as the driver routine; another that applies a rule; a predicate ruleset that decides how to prove a hypothesis; and, finally, another predicate that asks the user to confirm a hypothesis.

Rule [1] in

To diagnose:

iterates through each of the possible hypotheses. On each iteration, it applies the rule that could prove that hypothesis--for simplicity we assume only one such rule for each of the initial hypotheses. Note that the **until** part of rule [1] is tested *after* the rule is applied, terminating rule [1] when a hypothesis is proved by the rule for which it is a conclusion.

The next ruleset,

To apply a rule:

shows that ANIMAL also assumes a three-valued logic system, i.e., a hypothesis can either be true, false, or unknown. There are three situations that can occur when applying a rule: (1) each of the rule's preconditions can be true, in which case its conclusion is true; (2) all of the rule's preconditions are false (i.e., not unknown), in which case its conclusion is false; or (3) some of its preconditions are unknown, in which case nothing can be said about the hypothesis.

The predicate ruleset,

To decide if a hypothesis is true:

is straightforward up to rule [4]. Rule [4] addresses the situation of whether **the hypothesis** is the conclusion of some rule. If it is, then it could be the conclusion of several rules, in which case each is applied until **the hypothesis** is either confirmed or denied. If not a conclusion, then the user is asked to confirm. Control enters the **default** block if the user denies the hypothesis or labels it as unknown.

Finally, the predicate ruleset,

To decide if a hypothesis is confirmed:

uses the **query for** system generator to convert **the hypothesis** into a yes/no question. The user is prompted with this question until he answers **yes**, **no**, or **?**, confirming, denying, or labeling as unknown **the hypothesis**, respectively.

The interactions seen below demonstrate how ANIMAL works when trying to identify the user's choice, which in this case is a cheetah. Note that **diagnose** is called with the database **conclusions** active, thus allowing us to see which conclusions are made by ANIMAL.

<2> Activate conclusions.

<3> ?

[CONCLUSIONS Database]

<4> display every hypothesis.

'ANIMAL IS AN ALBATROSS'

'ANIMAL IS A PENGUIN'

'ANIMAL IS AN OSTRICH'

'ANIMAL IS A ZEBRA'

'ANIMAL IS A GIRAFFE'

'ANIMAL IS A TIGER'

'ANIMAL IS A CHEETAH'

<5> Diagnose.

Q: DOES ANIMAL HAVE FEATHERS?

A: no

Q: DOES ANIMAL HAVE HAIR?

A: ?

Q: DOES ANIMAL GIVE MILK?

A: yes

Q: DOES ANIMAL HAVE HOOFS?

A: no

Q: DOES ANIMAL EAT MEAT?

A: ?

Q: DOES ANIMAL HAVE ITS EYES POINTED AHEAD?

A: yes

Q: DOES ANIMAL HAVE CLAWS?

A: yes

Q: DOES ANIMAL HAVE POINTED TEETH?

A: yes

Q: IS ANIMAL TAWNY COLORED?

A: yes

Q: DOES ANIMAL HAVE BLACK STRIPES?

A: no

Q: DOES ANIMAL HAVE DARK SPOTS?

A: yes

Hypothesis: 'ANIMAL IS A CHEETAH'

<6> ?

[CONCLUSIONS Database]

ANIMAL DOES GIVE MILK.
ANIMAL DOES HAVE DARK SPOTS.
ANIMAL DOES NOT HAVE BLACK STRIPES.
ANIMAL DOES HAVE POINTED TEETH.
ANIMAL DOES HAVE CLAWS.
ANIMAL DOES HAVE ITS EYES POINTED AHEAD.
ANIMAL DOES NOT HAVE HOOFS.
ANIMAL DOES NOT HAVE FEATHERS.
ANIMAL IS TAWNY COLORED.
'ANIMAL DOES EAT MEAT' IS UNKNOWN.
'ANIMAL DOES HAVE HAIR' IS UNKNOWN.
ANIMAL IS A CHEETAH.
ANIMAL IS NOT A TIGER.
ANIMAL IS A CARNIVORE.
ANIMAL IS NOT A GIRAFFE.
ANIMAL IS NOT A ZEBRA.
ANIMAL IS NOT AN UNGULATE.
ANIMAL IS A MAMMAL.
ANIMAL IS NOT AN OSTRICH.
ANIMAL IS NOT A PENGUIN.
ANIMAL IS NOT AN ALBATROSS.
ANIMAL IS NOT A BIRD.

APPENDIX B: ERROR MESSAGES

There are several types of errors that can be encountered during the course of using ROSIE. There are errors that can occur during the tokenization of a file or when parsing the results of tokenization, and there are errors that can occur at runtime, all but one of which is recoverable.¹ In each case, the error message printed provides an indication of the problem. Below is a list of the possible error messages (in boldface) each of which is followed by a short comment to help diagnose the problem.

PARSING AND TOKENIZATION ERRORS

The following error messages will be encountered during tokenization and parsing. Note that tokenization errors automatically abort the parsing task, sending control back to the top-level monitor. Parsing errors do not abort the parsing tasks. If a parsing error occurs when parsing a program file, ROSIE will print a message indicating where in the program file the erroneous code can be found.

Can't find anaphoric reference:

Parsing context

Encountered during parsing.

An anaphoric term (**that class noun**) or an anaphoric description (**such class noun**) references a description that is not processed before it in the rule currently being parsed.

See Section 7.5.

Discarding unexpected END STATEMENT.

Warning.

Encountered during parsing.

An end statement was encountered that did not terminate a ruleset. Discarded end statement will not appear in the **.txt** file when saved.

See Section 4.2.

¹A stack overflow, giving the error message "Computation depth limit exceeded", is currently the only nonrecoverable runtime error.

Discarding unexpected MONITOR DECLARATION.

Warning.

Encountered during parsing.

An execution monitor declaration can appear only once in a ruleset and only immediately before the first rule. Warning given when this declaration is found anywhere else. Discarded declaration will not appear in the `.txt` file when saved.

See Section 4.2.

Discarding unexpected PRIVATE DECLARATION.

Warning.

Encountered during parsing.

A private class declaration can appear only once in a ruleset and only immediately after the rule header. Warning given when this declaration is found anywhere else. Discarded declaration will not appear in the `.txt` file when saved.

See Section 4.2.

Has atomic formal parameters.

Encountered when parsing a system ruleset.

The formal parameters of a system ruleset body must be a list or NIL.

Illegal PATTERN VARIABLE specification:

Parsing context

Encountered during parsing.

A pattern variable can be specified only as a single token name, a description introduced by the function word **the** or an anaphoric term (i.e., introduced by **that**).

See Section 9.6.

Illegal specification of units:

Parsing context

Encountered during parsing.

The units of a unit constant must be atomic tokens combined under multiplication, division, or exponentiation, e.g.,

34 k*m/s^2

This error is called when these units are incorrectly combined.

See Section 9.3.

Inserting missing END STATEMENT.

Warning.

Encountered during parsing.

Called when processing a ruleset and encounters the start of a new ruleset or the last file rule before the terminating end statement. Adds end statement automatically.

Not a lisp lambda-form.

Encountered when parsing a system ruleset.

Expression read after system ruleset header (using the LISP `read` function) is not a LISP *lambda-form*.

See Section 4.4.

Obsolete use of EXECUTE ACTION:

Parsing context

Warning.

Encountered during parsing.

Earlier ROSIE's provided the *execute* actions, `call` and `go`. In ROSIE 3.0 these have been subsumed by the procedure action type. While ROSIE 3.0 still supports the *execute* actions, they may not be supported in the future and should be removed from your code.

QUANTIFIED TERMS may not appear in BREAK COMMANDS:

Parsing context

Encountered during parsing.

A quantified descriptive term or iterative term was found to occupy the position of `<term>` in a break command of the form

Produce <term>!

Such terms cannot be used in this position.

QUANTIFIED TERMS may not appear in PRIVATE DECLARATIONS:*Parsing context*

Encountered during parsing.

A quantified descriptive term or iterative term was found to occupy the position of <term> in a private class declaration of the form

Private: . . . <class> (initially <term>)

Such terms cannot be used in this position.

Replacing NLAMBDA with LAMBDA.**Warning.**

Encountered when parsing a system ruleset.

In earlier releases of ROSIE (Interlisp), a system ruleset body was required to be a *spreading-NLAMBDA*, while in ROSIE 3.0, they are required to be a LISP *lambda form*. The difference is that one starts with the keyword NLAMBDA while the other LAMBDA.

This warning is called when a system ruleset body is found to start with an NLAMBDA. Assumes pre-ROSIE 3.0 system ruleset and automatically replaces with a LAMBDA. New ruleset will appear in the **.txt** file when saved.

Unexpected end of file.

Encountered during tokenization.

The tokenizer reached the end of input while in the process of scanning a file item, i.e., the file item was improperly terminated.

Unmatched left parenthesis or missing body after:*System ruleset header*

Encountered during tokenization.

Unable to read body (a LISP *lambda form*) corresponding to *system ruleset header* using the LISP **read** function. Probably caused by mismatched parentheses.

*Prep used twice in a prepositional phrase:
Parsing context*

Encounter during parsing.

The same preposition (*prep*) appears twice in the chain of preposition/term pairs associated with a procedure, proposition, description, or ruleset header.

RUNTIME ERRORS

The following errors will be encountered in normal runtime operations. Errors that are particular to a small set of rulesets from the system support library are noted as such. All but one of the errors seen here are recoverable. When called, a recoverable error throws control into a break loop, from which the error can be corrected and computations continued. The nonrecoverable error aborts computations and throws control back to the top-level monitor.

NOTE: The following error messages (starting **Bad argument type . . .**) are called by system rulesets when an argument is not of the type expected. The type expectation that was violated is specified as (**not TYPE**). The argument causing the error is given in italics after the error message.

Bad argument type to ACTIVATE (not NAME):
The database

Called by:

To activate *a database*

Bad argument type to ADD (not NAME):
The database

Called by:

To add *a proposition* to a database

Bad argument type to ADD (not PROPOSITION):
The proposition

Called by:

To add *a proposition* to a database

Bad argument type to ASSERT (not NAME):

The database

Called by:

To assert *a proposition* in *a database*

Bad argument type to ASSERT (not PROPOSITION):

The proposition

Called by:

To assert *a proposition* [in *a database*]

Bad argument type to CLEAR (not NAME):

The database

Called by:

To clear *a database*

Bad argument type to CONCATENATION (not TUPLE):

The tuple

Called by:

To generate the concatenation of *a tuple* with *a tuple*

Bad argument type to COPY (not NAME):

The database

Called by:

To copy from *a database*

To copy to *a database*

Bad argument type to DECREMENT (not DESCRIPTION):

The description

Called by:

To decrement *a description* [by *an amount*] [in *a database*]

Bad argument type to DECREMENT (not NAME):

The database

Called by:

To decrement *a description* by *an amount* in *a database*

Bad argument type to DECREMENT (not NUMBER):

The amount

Called by:

To decrement *a description* [by *an amount*] [in *a database*]

Bad argument type to DENY (not NAME):

The database

Called by:

To deny *a proposition* from *a database*

Bad argument type to DENY (not PROPOSITION):

The proposition

Called by:

To deny *a proposition* [from *a database*]

Bad argument type to DESCRIBE (not NAME):

The database

Called by:

To describe *an element* in *a database*

Bad argument type to DUMP (not NAME):

The database

Called by:

To dump *a database* as *a file*

Bad argument type to EMPTY (not TUPLE):

The tuple

Called by:

To decide if *a tuple* is empty

Bad argument type to EXECUTE (not PROCEDURE):

The procedure

Called by:

To execute *a procedure*

Bad argument type to FALSE (not NAME):

The database

Called by:

Before denying *a proposition* is [not] false in *a database*

Before asserting *a proposition* is [not] false in *a database*

To decide if *a proposition* is false in *a database*

Bad argument type to FALSE (not PROPOSITION):

The proposition

Called by:

Before denying *a proposition* is [not] false in *a database*

Before asserting *a proposition* is [not] false in *a database*

To decide if *a proposition* is false in *a database*

Bad argument type to FIRST MEMBER (not TUPLE):

The tuple

Called by:

To generate the first member of *a tuple*

Bad argument type to FIX (not INTEGER):

The line

Called by:

To fix *a line*

Bad argument type to FORGET (not NAME):

The database

Called by:

To forget about *an element* in *a database*

Bad argument type to INCREMENT (not DESCRIPTION):

The description

Called by:

To increment *a description* [by *an amount*] [in *a database*]

Bad argument type to INCREMENT (not NAME):

The database

Called by:

To increment *a description* by *an amount* in *a database*

Bad argument type to INCREMENT (not NUMBER):

The amount

Called by:

To increment *a description* [by *an amount*] [in *a database*]

Bad argument type to INSTANCE (not NAME):

The database

Called by:

To generate an instance of *an element* in *a database*

Bad argument type to INSTANTIATE (not DESCRIPTION):

The description

Called by:

To instantiate *a description* to *an element* [in *a database*]

Bad argument type to INSTANTIATE (not NAME):

The database

Called by:

To instantiate *a description* to *an element* in *a database*

Bad argument type to INTEGER (not INTEGER):

The lower bound or The upper bound or The step

Called by:

To generate an integer from *a lower bound* to *an upper bound* [by *a step*]

Bad argument type to LAST MEMBER (not TUPLE):

The tuple

Called by:

To generate the last member of *a tuple*

Bad argument type to MEMBER (not INTEGER):

The position

Called by:

To decide if *an element* is a member of *a tuple* at *a position*
To decide if *an element* is a member of *a tuple* from *a position*
To generate the member of *a tuple* at *a position*
To generate a member of *a tuple* from *a position*

Bad argument type to MEMBER (not TUPLE):

The tuple

Called by:

To decide if *an element* is a member of *a tuple* [at *a position*]
To decide if *an element* is a member of *a tuple* from *a position*
To generate the member of *a tuple* [at *a position*]
To generate a member of *a tuple* from *a position*

Bad argument type to NEGATED (not PROPOSITION):

The proposition

Called by:

To decide if *a proposition* is negated

Bad argument type to NUMBER (not NUMBER):

The lower bound or the upper bound or the step

Called by:

To generate a number from *a lower bound* to *an upper bound* [by *a step*]

Bad argument type to PRINT (not NAME):

The database

Called by:

To print *a name as a string*

Bad argument type to PROPOSITION (not NAME):

The database

Called by:

To generate an affirmed proposition from *a database*

Bad argument type to PROVABLY (not PROPOSITION):

The proposition

Called by:

Before denying *a proposition* is [not] provably *true/false*
Before asserting *a proposition* is [not] provably *true/false*
To decide if *a proposition* is provably *true/false*

Bad argument type to QUERY (not PROPOSITION):

The proposition

Called by:

To generate a query for *a proposition*

Bad argument type to REDO (not INTEGER):

The line

Called by:

To redo *a line* [thru *a line*] [for *N* times]

Bad argument type to REDO (not NUMBER):

N times argument

Called by:

To redo *a line* [thru *a line*] for *N* times

Bad argument type to REMOVE (not NAME):

The database

Called by:

To remove *a proposition* from *a database*

Bad argument type to REMOVE (not PROPOSITION):

The proposition

Called by:

To remove *a proposition* from *a database*

Bad argument type to RESTORE (not NAME):

The database

Called by:

To restore *a file* to *a database*

Bad argument type to REVERSE (not TUPLE):

The tuple

Called by:

To generate the reverse of *a tuple*

Bad argument type to SECOND MEMBER (not TUPLE):

The tuple

Called by:

To generate the second member of *a tuple*

Bad argument type to SET (not NAME):

The switch

Called by:

To decide if *a switch* is set

Bad argument type to SHOW (not NAME):

The database

Called by:

To show *a database*

Bad argument type to SORT (not TUPLE):
The tuple

Called by:

To sort *a tuple* in *an order*

Bad argument type to SWAP (not NAME):
The database

Called by:

To swap in *a database*

Bad argument type to SWITCH (not NAME):
The switch

Called by:

To switch off *a switch*
To switch on *a switch*

Bad argument type to TAB (not INTEGER):
The column

Called by:

To tab to *a column* [*in a file*]

Bad argument type to TAIL (not INTEGER):
The position

Called by:

To generate the tail of *a tuple* at *a position*
To generate the tail of *a tuple* from *a position*

Bad argument type to TAIL (not TUPLE):
The tuple

Called by:

To generate the tail of *a tuple* [*at a position*]

To generate the tail of *a tuple* from *a position*

Bad argument type to TOGGLE (not NAME):

The switch

Called by:

To toggle off *a switch*
To toggle on *a switch*
To toggle *a switch*

Bad argument type to TRUE (not NAME):

The database

Called by:

Before denying *a proposition* is [not] true in *a database*
Before asserting *a proposition* is [not] true in *a database*
To decide if *a proposition* is true in *a database*

Bad argument type to TRUE (not PROPOSITION):

The proposition

Called by:

Before denying *a proposition* is [not] true in *a database*
Before asserting *a proposition* is [not] true in *a database*
To decide if *a proposition* is true in *a database*

Bad value from SYSTEM GENERATOR:

Value

Called when a generator ruleset defined as a system ruleset returns something (*value*) that is neither a LISP *atom* or *list*.

See Section 4.4.

CLASS ELEMENT returned from PRODUCE DEMON:

Element

Class elements cannot be returned from a produce demon.

Computation depth limit exceeded.

Nonrecoverable error.

Encountered at runtime.

Called when ruleset invocation stack exhausted (size: 120 frames).
Indicates infinite loop or poor program design or both.

CONCLUDE not inside PREDICATE.

The **conclude** procedure can be called only from a predicate ruleset.

See Section 4.2.

CONTINUE not inside DEMON.

The **continue** procedure can be called only from a demon.

See Section 4.3.

Can't close "OS:" channel.

Can't close "TTY:" channel.

Called by:

To close *a file*

when *file* is either "OS:" or "TTY:".

See Section 11.1.

Can't find program files for:

Filesegment

Called by any of the file package operations that try to load or rename a program file (*filesegment*) and can't find the .txt or .map files on disk.

See Chapter 13.

Defined as SYSTEM RULESET:

Filesegment

Called when trying to break, trace, or profile a rule inside of *filesegment* when it names a system ruleset.

See Section 14.1.

File already open:

The file

Called by:

To open *a file* for *input/output*

when *a channel to file* is already open.

See Section 11.1.

File not open for input:

The file

Called by:

To read *a string* from *a file*

when *file* is not open for input.

See Section 11.1.

File not open for output:

The file

Called by:

To send *a string* to *a file*

To tab to *a column* in *a file*

when *file* is not open for output.

See Section 11.1.

File not open:

The file

Called by:

To close *a file*

when *file* is not open.

See Section 11.1.

Filesegment already exists:

The filespec

Called by:

To build a *filespec*

when *filespec* names a program file that has already been noticed.

Filesegment not broken:

Filesegment

Called when trying to unbreak or untrace a ruleset (*filesegment*) that is not broken.

Filesegment not enabled:

Filesegment

Called when trying to break, trace, or profile a ruleset from some program file when that ruleset is not enabled. This is not the same as trying to break a ruleset that is not enabled when the program file of the ruleset is not given.

See Section 14.1.

Filesegment unknown to system:

Filesegment

Called from any of the file package and break package operations when *filesegment* is not noticed.

See Chapter 13.

Illegal argument to EVALUATE:

Timer argument

Called by:

To evaluate a *rule* [against timer]

when the *timer argument* is anything but **timer**.

Illegal argument to STOP:

Dribbling argument

Called by:

To stop dribbling

when the *dribbling argument* is anything but **dribbling**.

Illegal BOX width:

Element

Called from the **box** subpattern.

See Section 9.6.

Illegal comparison:

Element1 op element2

Called by one of the comparison operators (*op*) when the operands (*element1* and *element2*) cannot be compared under that operation.

See Section 9.3.

Illegal expression to EVALUATE:

The rule

Called by:

To evaluate *a rule* [against timer]

when *rule* cannot be parsed as a ROSIE rule.

Illegal I/O access to AVAILABLE:

Input argument

Called by:

To decide *a file* is available for input

when the *input argument* is anything but **input**.

Illegal I/O access to OPEN:

Input/output argument or read/write argument

Called by:

To decide if *a file* is open for *input/output*

To open *a file* for *input/output*

To open *a file* to *read/write*

when the *input/output argument* is anything but **input** or

output, or (in the case of the **is open** predicate)
input/output, or when the *read/write argument* is anything
but **read** or **write**.

Illegal I/O access to REDIRECT:

Input/output argument

Called by:

To redirect *input/output* [to a file]

when the *input/output argument* is anything but **input** or **output**.

Illegal operation:

Element1 op element2

Called by one of the arithmetic operators (*op*) when the
operands (*element1* and *element2*) can't be combined under
that operation.

See Section 9.3.

Illegal order to SORT:

The order

Called by:

To sort *a tuple* in *an order*

when *order* is anything but **ascending order**, **descending order**,
ascending pair order, or **descending pair order**.

Illegal truth value to CONCLUDE:

True/false argument

Called by:

To conclude *true/false*

when the *true/false argument* is anything but **true** or **false**.

Illegal truth value to PROVABLY:

True/false argument

Called by:

Before denying *an element* is [not] provably *true/false*
Before asserting *an element* is [not] provably *true/false*
To decide if *an element* is provably *true/false*

when the *true/false argument* is anything but **true** or **false**.

Illegal tuple to SORT:

The tuple

Called by:

To sort *a tuple* in *an order*

when *tuple* does not contain elements that can be sorted.

See Section 9.4.

Illegal unit of measure to [ARC]COSINE:

Radians argument

Called by:

To generate the [arc]cosine of *a number* in radians

when the *radians argument* is anything but **radians**.

Illegal unit of measure to [ARC]SINE:

Radians argument

Called by:

To generate the [arc]sine of *a number* in radians

when the *radians argument* is anything but **radians**.

Illegal unit of measure to [ARC]TANGENT:

Radians argument

Called by:

To generate the [arc]tangent of *a number* in radians

when the *radians argument* is anything but **radians**.

Index out of range in MEMBER:

The position

Called by:

- To decide if *an element* is a member of a tuple at a position
- To decide if *an element* is a member of a tuple from a position
- To generate the member of a tuple at a position
- To generate a member of a tuple from a position

when *position* is negative or larger than *tuple*.

Index out of range in TAIL:

The position

Called by:

- To generate the tail of a tuple at a position
- To generate the tail of a tuple from a position

when *position* is negative or larger than *tuple*.

Input from file won't match pattern:

The pattern

Called by:

- To read a pattern from a file

See Section 9.6.

Lexical error detected in:

Filename

Encountered during parsing.

Called when a lexical error occurred during tokenization of a file (*filename*).

No DRIBBLE FILE currently active.

Called by:

- To stop dribbling

No such file exists:

Filename

Called when attempting to open a file (*filename*) that cannot

be found on disk.

No such element exists:

THE description

Called when evaluating a simple descriptive term introduced by the function word **the** and no element can be generated from *description*.

See Section 8.4.

NOTE: When *description* is a call to a system generator, this error indicates that the arguments to the generator were bad. Unlike the other rulesets from the system support library, system generators do not normally call an error when passed bad arguments, rather they produce nothing. This is done to permit graceful error recovery, e.g., the predicate **there is a description** fails if *description* produces nothing, but does not call an error.

Not able to open file:

Filename

Called when a file (*filename*) exists but cannot be open.

Not coercible into a filename:

Element

Called by a ruleset that received *element* as a filename argument. *Element* could not be coerced into a filename.

See Section 11.1.

Nothing saved for line *N*.

Called by history facility when attempt is made to access monitor rule *N* and that rule is not one of the last 40 monitor rules seen.

See Chapter 3.

Only files may be renamed:

The source filespec or the target filespec

Called by:

To change a *source filespec* to a *target filespec*

when either *filespec* names a portion of a program file rather than a complete program file.

PRODUCE not inside GENERATOR.

The **produce** procedure can be called only from a generator ruleset or a produce demon.

See Section 4.2.3.5.

Pattern not coercible to string:

Pattern

Called when attempting to coerce *pattern* into a string and *pattern* describes a language of more than one string.

See Section 9.6.

Procedure not defined:

Procedure

Called when attempts to invoke *procedure* that currently is not enabled.

Syntax error detected in:

Filesegment

Called by break package when trying to break, trace, or profile a ruleset rule (*filesegment*) when the ruleset contains a syntax error.

See Chapter 14.

Terminal input won't match pattern:

The pattern

Called by:

To read a *pattern*

See Section 9.6.

Unbound ANAPHORIC TERM:

THAT class noun

Called when evaluating an anaphoric term (*that class noun*) that references an unbound description variable.

See Section 7.3.

Unbound RULE VARIABLE:

Variable

Called when evaluating a rule variable (*variable*) that references an unbound description variable.

See Section 7.3.

APPENDIX C: SYSTEM SWITCHES

ROSIE supports a small number of *system switches* to control certain aspects of system behavior (these switches are implemented as LISP variables whose values are `T` when *on* and `NIL` when *off*). Some switches are supported to make ROSIE 3.0 act like earlier versions of ROSIE and others simply to suppress noncritical features that the user may not like.

The system switches include:

\$AUTOQUERYFLG -- (default setting: *off*)

This switch controls ROSIE's actions when it can neither prove nor disprove the truth or falsity of a proposition, i.e., the action if, after examining the database and rulesets, the truth value of a proposition is still unknown.

When this switch is *on*, ROSIE will look for the predicate ruleset,

To decide if a proposition is confirmed:

If such a ruleset exists, it will be applied to the positive form of the proposition. The conclusion of the ruleset decides the truth or falsity of this form, which then decides the truth of the target proposition.

ROSIE provides a default query mode predicate defined as

To decide if a proposition is confirmed:

Private: a reply.

Execute cyclically.

[1] Send "{cr}{the query for the proposition} ".

```
[2] Read "{anything (bind to the reply)}{cr}".
```

[3] Select the uppercase of the reply:

```

<"YES"> assert the proposition is provably true
          and conclude true;

```

```
<"NO"> assert the proposition is provably false
        and conclude false;
```

```
<"> return;
```

default: send "{cr}Please respond YES or NO{cr}".

End.

To see how this works, consider the following sample session

```
<2> ?  
[ GLOBAL Database ]  
  
<3> If John is a man, display yes.  
<4> Switch on $AUTOQUERYFLG.  
<5> Redo 3.  
  
IS JOHN A MAN? y  
  
Please respond YES or NO  
  
IS JOHN A MAN? YES  
YES  
<6> ?  
[ GLOBAL Database ]  
    JOHN IS A MAN.  
  
<7> Redo 3.  
YES  
  
<8> If John does not love Mary, display yes.  
  
DOES JOHN LOVE MARY? YES  
<9> ?  
[ GLOBAL Database ]  
    JOHN DOES LOVE MARY.  
    JOHN IS A MAN.
```

The auto-query mechanism allows ROSIE to build up its database by consulting the user. While this mechanism is not always appropriate, it is extremely useful for diagnostic tasks.

```
$COMPRULESETS  -- (default setting: off)  
$EXPDRULESETS  -- (default setting: on)
```

These two switches apply when a ruleset is defined (*enabled*) in a ROSIE session.

When the **\$COMPRULESETS** switch is *on*, the definition of a ruleset will be compiled in core when defined.

When the **\$EXPDRULESETS** switch is *on*, the definition of a ruleset will be optimized (through a series of macro expansions) when defined.

When either of these flags is set, load time of a program file will be increased about ten times. For program files containing a dozen or more rulesets, this means an increase from a matter of seconds to half a minute or more, but the compiled and optimized (compiling also optimizes) definitions run much faster.

\$EXTENDSEARCH -- (default setting: *on*)

This switch controls the manner in which ROSIE conducts its search of the physical database--yet another aspect of ROSIE 3.0 that differs from earlier ROSIEs.

In earlier releases, the physical database structure was two-tiered. When searching the physical database, ROSIE would first examine the private database then the active. It would examine only the global database when active.

The structure of the physical database in ROSIE 3.0 is three-tiered. ROSIE first consults the private database (which has been reduced to a much restricted form), then the active database, and then the global database. If the global database is also the active database, it will only be examined once.

When this switch is *off*, ROSIE returns to a two-tiered database structure. The following sample session illustrates the difference.

```
<2> Assert each of Jim, Jack and John is a man.
<3> ?
[ GLOBAL Database ]
    JOHN IS A MAN.
    JACK IS A MAN.
    JIM IS A MAN.

<4> Activate tmp.
<5> ?
[ TMP Database ]

<6> Display every man.
JOHN
JACK
JIM
<7> If John is a man, display yes, otherwise display no.
YES
<8> Switch off $EXTENDSEARCH.
<9> Display every man.
<10> If John is a man, display yes, otherwise display no.
NO
```

\$MIXPRINTMODE -- (default setting: *off*)

When ROSIE 3.0 displays things like the evaluation name of an element, e.g.,

```
<19> Display 'John is a man'.
'JOHN IS A MAN'
```

the contents of a database, e.g.,

<20> Assert each of Jim, Jack and John is a man.

<21> ?

[GLOBAL Database]

JOHN IS A MAN.

JACK IS A MAN.

JIM IS A MAN.

or coerces an element into a string, e.g.,

<22> Display the string from {the man}.

"JOHN"

it does so putting all characters (except those within string tokens) into uppercase. In earlier releases, such output was done in mixed case, e.g.,

<23> Switch on \$MIXPRINTMODE.

<24> Redo 19 thru 22.

'JOHN is a man'

[GLOBAL Database]

JOHN is a man.

JACK is a man.

JIM is a man.

"JOHN"

Fixed syntactic constructs, such as prepositions, appeared in lowercase while arbitrary arguments appeared in uppercase. Mixed case is helpful when learning to distinguish between such constructs, but is undesirable in a finished system.

When **\$MIXPRINTMODE** is *on*, output appears as it would in earlier releases of ROSIE. When *off*, output is in uppercase.

\$PRETTYFORMAT -- (default setting: *off*)

This switch also controls the format of output. When *on*, output will appear in lowercase (even if **\$MIXPRINTMODE** is *on*). Further, strings will appear without surrounding double quotes, the intentional elements without single quotes, and the first nonseparator character of the string will be capitalized, e.g.,

<25> Switch on \$PRETTYFORMAT.

<26> Display 'John is a man'.

John is a man

The **print** procedure is equivalent to the **send** procedure when **\$PRETTYFORMAT** is *on* (see Chapter 11).

This switch also allows the user to control the format of explicitly named tokens, working in conjunction with the **print as** procedure.

The **print as** procedure takes two arguments, one a name element and the other a string. When the **\$PRETTYFORMAT** is *on*, instances of the name will be output as the given string, e.g.,

```
<27> Print John Brown as "John Brown".
<28> Display 'John Brown is a man'.
John Brown is a man
```

\$PRINTMSGs -- (default setting: *on*)

This switch controls whether miscellaneous system messages are output. For instance, most of the file package commands output messages that trace their operation. While such messages are not necessary for system performance, they do provide a sense of security that comes from knowing what's happening. In a finished ROSIE expert system, such messages may appear irrelevant. They can be disabled by switching off **\$PRINTMSGs**.

\$REMOVEDUPLs -- (default setting: *off*)

This switch deals with the generation of elements from a description. It enables a feature found in older releases of ROSIE but that is not the standard for ROSIE 3.0.

Descriptions are thought of as implicitly naming a set of elements. Further, descriptions can be used as generators to produce these elements in sequence. In an attempt to ensure the purity of the notion that descriptions named sets, earlier ROSIEs always removed duplicate elements from the sequence produced by a description, e.g., in ROSIE (Version 2), the following would occur

```
<10> Display every member of <1,2,2,3,3,3>.
1
2
3
```

while in ROSIE 3.0,

```
<10> Display every member of <1,2,2,3,3,3>.
1
2
2
3
3
3
```

This change was not the result of some great conceptual insight, but prompted purely by pragmatics. The test to ensure that an element wasn't produced twice turned generation of elements into an $O(n^2)$ process.

Thus, while this feature ensured set purity, set purity was rarely an issue, often an annoyance, and greatly hindered system performance.

The old mechanism was not removed from ROSIE 3.0, merely disabled. When **\$REMOVEDUPLS** is *on*, this mechanism is reenabled.

OPERATIONS ON SYSTEM SWITCHES

The operations for turning switches *on* and *off* as well as for checking their setting are defined as follows:

switch on *a switch*

switch off *a switch*

Respectively, enables or disables *switch*, which is one of ROSIE's system switches.

<15> Switch on \$MIXPRINTMODE.

<16> ?

[GLOBAL Database]

SARA is a woman.

MARY is a woman.

JOHN is a man.

JACK is a man.

JIM is a man.

toggle on *a switch*

toggle off *a switch*

Like **switch on/off** except that, if executed in a ruleset, *switch* reverts to its original setting when the ruleset terminates.

toggle *a switch*

If *switch* is *on*, turns it *off*, otherwise turns it *on*.

a switch is set

Concludes true if *switch* is *on*, concludes false otherwise, e.g.,

<17> If \$MIXPRINTMODE is set, display yes.

YES

info switches

Lists the setting of all system switches, e.g.,

<18> Info switches.

```
$AUTOQUERYFLG is off
$COMPRULESETS is off
$EXPDRULESETS is on
$EXTENDSEARCH is on
$MIXPRINTMODE is on
$PRETTYFORMAT is off
$PRINTMSGs is on
$REMOVEDUPLS is off
```

REFERENCES

- Anderson, R. H., and J. J. Gillogly, *RAND Intelligent Terminal Agent (RITA): Design Philosophy*, The RAND Corporation, R-1809-ARPA, 1976.
- Anderson, R. H., M. Gallegos, J. J. Gillogly, R. B. Greenberg, and R. V. Villanueva, *RITA Reference Manual*, The RAND Corporation, R-1808-ARPA, 1977.
- Beebe, H. M., H. S. Goodman, G. L. Henry, and D. S. Snell, "The Adept Workstation: A Knowledge-Based System for Combat Intelligence Analysis," *Proceedings of the Seventh MIT/ONR Workshop on C3 Systems*, Massachusetts Institute of Technology, Cambridge, MA, 1984.
- Callero, M., D. A. Waterman, and J. R. Kipps, *TATR: A Prototype Expert System for Tactical Air Targeting*, The RAND Corporation, R-3096-ARPA, 1984.
- Fain, J., D. Gorlin, F. Hayes-Roth, S. Rosenschein, H. Sowizral, and D. A. Waterman, *The ROSIE Language Reference Manual*, The RAND Corporation, N-1647-ARPA, 1981.
- Fain, J., F. Hayes-Roth, H. Sowizral, and D. A. Waterman, *Programming in ROSIE: An Introduction by Means of Examples*, The RAND Corporation, N-1646-ARPA, 1982.
- Forgy, C. L., *The OPS5 User's Manual*, Technical Report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1981.
- Galway, W., M. L. Griss, B. Morrison, and B. Othmer, *The Portable Standard LISP User Manual*, The Utah Symbolic Computation Group, University of Utah, Salt Lake City, UT, 1984.
- Hayes-Roth, F., D. A. Waterman, and D. Lenat (eds.), *Building Expert Systems*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1983.
- Hayes-Roth, F., D. Gorlin, S. Rosenschein, H. Sowizral, and D. A. Waterman, *Rationale and Motivation for ROSIE*, The RAND Corporation, N-1648-ARPA, 1981.
- Irons, E. T., "Syntax Graphs and Fast Context-Free Parsing," Research Report 71-1, Yale University, New Haven, CT, 1971.
- Kowalski, R., "Algorithm = Logic + Control," *Communications of the ACM*, Vol. 22, No. 7, 1979.

- Kruppenbacher, T. A., "The Application of Artificial Intelligence to Contract Management," Masters thesis, Department of Civil, Environmental and Architectural Engineering, University of Colorado, Boulder, CO, 1984.
- McDermott, J., and C. Forgy, "Production System Conflict Resolution Strategies," D. A. Waterman and F. Hayes-Roth (eds.), in *Pattern-Directed Inference Systems*, Academic Press, New York, NY, 1978.
- Pagan, F. G., *Formal Specification of Programming Languages: A Panoramic Primer*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
- Paul, J., D. A. Waterman, and M. A. Peterson, "SAL: An Expert System for Evaluating Asbestos Claims," *Proceedings of the First Australian Artificial Intelligence Congress*, Melbourne, 1986.
- Sowizral, H. A., and J. R. Kipps, *ROSIE: A Programming Environment for Expert Systems*, The RAND Corporation, R-3246-ARPA, 1985.
- Teitelman, W. et al., *Interlisp Reference Manual*, (3d rev.) Xerox Palo Alto Research Center, Palo Alto, CA, 1978.
- Tomita, M., "An Efficient Context-free Parsing Algorithm for Natural Languages," *Proceedings of Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pp. 756-764, Los Angeles, CA, 1985.
- van Melle, W., E. H. Shortliffe, and B. G. Buchanan, "EMYCIN: A Domain-independent System That Aids in Constructing Knowledge-Based Consultation Programs," *Machine Intelligence, Infotech State of the Art*, Report 9, No. 3, 1981.
- Waterman, D. A., J. Paul, B. Florman, and J. R. Kipps, *An Explanation Facility for the ROSIE Knowledge Engineering Language*, The RAND Corporation, R-3406-ARPA, 1986.
- Waterman, D. A., and M. A. Peterson, *Models of Legal Decisionmaking*, The RAND Corporation, R-2717-ICJ, 1981.
- Waterman, D. A., R. H. Anderson, F. Hayes-Roth, P. Klahr, G. Martins, and S. J. Rosenschein, *Design of a Rule-Oriented System for Implementing Expertise*, The RAND Corporation, N-1158-ARPA, 1979.
- Winston, P.H., *Artificial Intelligence*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1979.

INDEX

\$AUTOQUERYFLG 230; 337
 \$COMPRULESETS 338
 \$EXPDRULESETS 338
 \$EXTENDSEARCH 339
 \$MIXPRINTMODE 339
 \$PRETTYFORMAT 176; 248; 340
 \$PRINTMSGs 341
 \$REMOVEDUPLS 341
 \$ROSIEEDITOR 49; 264
 cmp files 253
 db files (see *dump* and *restore*) 233
 map files 253
 rosie-ed 49
 rosierc 43; 264
 text files 255
 txt files 253
 <ctrl>C 56; 251
 <ctrl>D 64
 <ctrl>Z 58
 <atom> 27
 <integer> 27
 <number> 27
 <string> 27
 <CONTINUE> 87
 <FALSE> 87
 <TRUE> 87
 **
 *a number ** an integer* 164
 a number ^ an integer 164
 *
 *a number * a number* 164
 +
 a number + a number 164
 -
 a number - a number 164
 /
 a number / a number 164
 =
 an element [~]= an element 155
 <term> [~]= <term> 111; 151
 <term> [~]<[=] <term> 111
 <term> [~]>[=] <term> 111
 <
 a number [~]<[=] a number 165
 <term> [~]<[=] <term> 111
 >

- a number* [~]>[=] *a number* 165
- <term>* [~]>[=] *<term>* 111
- ?
- <integer>?* 59
- <name element>?* 59; 233
- ? 59; 233
- ?? 59
- private?* 273
- ,
- { *<subpat>* [, *<subpat>]** } 181
- |
- { *<subpat>* [| *<subpat>]** } 182
- a new*
- a new* *<description>* 140
- a/an*
- (| *a* | *an* |) *<description>* 140
- absolute value*
- the absolute value of a number* 165
- action blocks* 89
- actions* 89
 - conditional actions 95
 - conditional blocks 96
 - database actions 93; 220
 - execute actions 89
 - iterative actions 100
 - procedures 93
 - types of 89
- activate*
- activate* [*a database*] 232
- active database* 221
- active database*
- the active database* 232
- add*
- add a proposition to a database* 212; 235
- addition operator (+)* 164
- adjectives vs class compounds* 131
- adjoin*
- adjoin* *<subpat>* [, *<subpat>]** 185
- affirmed proposition*
- an affirmed proposition [from a database]* 212; 234
- affirmed propositions* 219
- alphanumeric*
- [non]alphanumeric[s] [[not] in *<term>*] 186
- alternate database*
- an alternate database* 233
- alternate databases* 221
- ambiguity* 38
- anaphoric descriptions* 127
- anaphoric reference* 126; 144
 - resolving ambiguity 128
- anaphoric terms* 126; 144

- and*
 - <action> and <action block>** 67
 - <disjunct> [, and <disjunct>]*** 105
 - <primary> and <conjunct>** 105
- ANIMAL 305
- antilog*
 - the [anti]log of a number** 166
- any*
 - any <description>** 201
- anything*
 - anything** 188
- apostrophe *s*
 - possessive case 139
- append*
 - append a file to a file** 250
- arccosine*
 - the [arc]cosine of a number [in radians]** 166
- arcsine*
 - the [arc]sine of a number [in radians]** 166
- arctangent*
 - the [arc]tangent of a number [in radians]** 166
- argument passing 78
- argument*
 - an argument of an element** 156
- arithmetic operators 26; 135
- arithmetic terms 135
- ASCII characters
 - generation of (see *charcode*) 191
- assert demons 82
- assert events 82; 94; 228
- assert*
 - assert a proposition [in a database]** 212; 235
 - assert <proposition> [and <proposition>]*** 94; 234
- asserting propositions 228
- assignment operator (see *let*) 94
- associativity 38
 - of action blocks 91
 - of arithmetic operators 137
 - of conditional actions 96
 - of conditional blocks 98
 - of conditions 106
 - of iterative actions 102
 - of unit constants 162
- atomic tokens 21
- auto-query mode 230
- available*
 - a file is available for input** 248
- backspace*
 - backspace[s]** 190
 - bs** 190
- backward reasoner in ROSIE 305

- be and do*
 - auxiliary forms of 26
- before*
 - before asserting** <pred form>: 81
 - before denying** <pred form>: 81
 - before executing** <proc form>: 81
 - before generating** <genr form>: 81
 - before producing** <genr form>: 81
 - before testing** [if] <pred form>: 81
- blank*
 - blank[s]** 190
 - [non]blank[s] [[not] in <term>]** 186
- BNF
 - lexical 27
 - linguistic 28
- boolean connectors 105
- both*
 - both <term> [,] and <term>** 147
- bottom*
 - bottom!** 273
- box*
 - box <subpat> to width <term>** 183
- break characters 18; 26
- break commands 22; 270
 - bottom!** 273
 - conclude false!** 272
 - conclude true!** 272
 - down!** 272
 - edit [(| ruleset | <integer> |)]!** 271
 - eval!** 270
 - help!** 273
 - list [(| ruleset | <integer> |)]!** 271
 - pop!** 273
 - produce an element!** 272
 - quit!** 273
 - result!** 271
 - resume [(| ruleset | <integer> |)]!** 271
 - return!** 271
 - top!** 273
 - trace!** 272
 - up!** 272
- break loops 56; 270
- break package 267
 - break facility 270
 - profile facility 278
 - trace facility 268
 - unbreaking 279
- break*
 - break [*a filespec*]** 280
- bs*
 - bs** 190

- build*
 - build a filespec** 260
- call* 89
- cardinality operators 112
- ceiling*
 - the ceiling of a number** 166
- center justify*
 - center justify** <subpat> [<dimen>] 184
 - CJ** [<term>] [**by** <term>] : <subpat> 184
- change*
 - change a filespec to a filespec** 262
- channel*
 - an open channel** 247
 - the OS channel** 247
 - the standard input channel** 247
 - the standard output channel** 247
 - the TTY channel** 247
- channels 241
 - closing 241
 - opening 241
 - operations on 246
 - OS channel 243
 - reading from 244
 - sending to 244
 - standard I/O channels 242
 - TTY channels 242
- character classes 18
- character codes
 - generation of 191
 - recognition of (see *codes*) 189
- characters*
 - character[s]** [[**not**] in <term>] 186
- charcode*
 - charcode** <term> 191
- choose*
 - choose situation:** [*selector*[;]]* [**default:** <action block>[;]] 97
- class elements 201
 - potential pitfalls 203
- class membership 108
 - generating 118
 - pitfalls 119
 - testing 116
- class nouns 116
- class relations 109
- class*
 - an element is a class** 154
 - the class from a string** 175
- classes 116
 - compounds vs adjectives 131
 - with relative clause 120
- clear*

- clear a database** 233
 - clear database** 233
- close*
 - close a file** 246
 - close everything** 246
- closing I/O channels 241
- codes*
 - codes (<integer> [, <integer>]*)** 189
- comma blocks 91
- comma-and 105
- comma-or 105
- comment characters 19
- comments 22
- communicating with the operating system 243
- comparison operators 111
- compile*
 - compile a filespec** 261
- compiling program files 254
- complement of a proposition 110
- complementation 108
- concatenation*
 - the concatenation of a tuple with a tuple** 169
- concatenation
 - in patterns 181
- conclude*
 - conclude false** 81
 - conclude false!** 272
 - conclude true** 81
 - conclude true!** 272
- conditional actions 90; 95
- conditional blocks 90; 96
- conditions 105
- confirm*
 - a proposition is confirmed** 230; 337
- conjunction
 - in conditions 105
 - in patterns 181
 - in relative clauses 121
- constant*
 - an element is a labeled constant** 164
 - an element is a unit constant** 164
- containing*
 - the tuple containing each <description>** 169
- continue*
 - continue** 81
- contradictory assertions 220
- control*
 - control <term>** 191
 - [non]control[s] [{not} in <term>]** 186
- copy*
 - a copy of an element** 156

- copy a file to a file** 249
 - copy a filespec after a filespec** 263
 - copy a filespec before a filespec** 263
 - copy from a database** 234
 - copy to a database** 234
- cosine**
 - the [arc]cosine of a number [in radians]** 166
- CR**
 - CR[s]** 189
- create**
 - create** <a/an> <description> 95; 235
- cyclic monitor** 72
- data types (see elements)** 134
- database actions** 90; 93; 220
- database**
 - a database** 232
 - an alternate database** 233
 - the active database** 232
- databases** 219
 - accessing** 223
 - activating** 221
 - active database** 221
 - alternate** 221
 - creating** 221
 - global database** 221
 - naming** 221
 - operations on** 232
 - physical** 219
 - private database** 221
 - virtual database** 224
- deactivate**
 - deactivate** 232
- debugging aids** 267
- declarations** 21
- decode**
 - decode a filespec** 263
- decrement**
 - decrement a description [by a number] [in a database]** 210; 237
- delete**
 - delete a file** 64; 250
- demo programs** 283
 - ANIMAL** 305
 - FORTUNE** 285
 - POIROT** 295
- demons** 81
 - assert** 82
 - deny** 82
 - generate** 82
 - generator** 83
 - procedural** 82
 - produce** 83

- test 82
- types of 82
- deny demons 82
- deny events 82; 94; 229
- deny
 - deny a proposition** [from a database] 213; 235
 - deny** <proposition> [and <proposition>]* 94; 235
- denying propositions 228
- deparse
 - deparse a filespec** 262
- describe
 - describe an element** [in a database] 61; 234
- description variables 125
- description
 - an element is a description** 155
 - the description from a string** 175
- descriptions 115
 - anaphoric 127
 - asserting members 130
 - denying members 130
 - description variables 125
 - generating members 129
 - testing membership 129
- descriptive terms 137
 - quantified 141
 - simple 138
- digit
 - [non]digit[s] [[not] in <term>] 186
- digits 19
- disable
 - disable a filespec** 264
- disabling program files 254
- disambiguation 38
- disjunction
 - in conditions 105
 - in patterns 182
 - in relative clauses 121
- display
 - display an element** 61; 248
- division operator (/) 164
- do
 - do nothing** 93
- down
 - down!** 272
- dribble
 - dribble to a file** 61; 250
 - stop dribbling** 250
- dskin
 - dskin a file** 64; 250
- dump
 - dump [a database] as a file** 233

- each of*
 - each of** <term> [, <term>]* [,] and <term> 147
- edit*
 - edit a filespec** 264
 - edit** [(| **ruleset** | <integer> |)]! 271
- editing program files 254; 256
- EDITOR (*unix shell variable*) 49; 264
- either*
 - either** <term> [, <term>]* [,} or <term> 146
- element type*
 - the element type of an element** 155
- element*
 - the element from a string** 174
- elements 149
 - as terms 134
 - class elements 201
 - equivalence vs. equality 151
 - evaluation names 151
 - filesegments 199
 - intentional 149
 - intentional descriptions 207
 - intentional procedures 217
 - intentional propositions 211
 - labeled constants 163
 - names 159
 - numbers 161
 - operations on 153
 - patterns 177
 - simple 149
 - simple numbers 161
 - strings 171
 - tuples 167
 - types of 149; 155
 - unit constants 162
- embedded control structures 305
- empty*
 - a tuple is empty** 167
- enable*
 - enable a filespec** 261
- enabling program files 254
- end statement 73
- end*
 - end** 190
 - end.** 73
- entering LISP (see *lisp*) 64
- EOL
 - EOL[s]** 190
- equal*
 - an element is equal to an element** 155
 - <term> is [not] equal to <term>** 111; 151
- equivalence vs. equality 151

- erase*
 - erase a filespec** 263
- error demon 85; 251
- error messages 313
- error*
 - <a string, a filesegment> is an error** 252
- errors 56; 251
 - error messages 313
 - nonrecoverable 251
 - parsing 313
 - recoverable 251
 - recovery 270
 - runtime 317
 - scanning 313
 - tokenization 313
- esc*
 - esc** 190
- escape characters 19
- escape*
 - esc** 190
 - escape[s]** 190
- eval*
 - eval!** 271
- evaluate*
 - evaluate a string {against timer}** 175
- evaluation names 151
- event-driven program control 81
- events
 - and demons 81
 - assert 82; 94; 228
 - continuing (see *continue*) 81
 - deny 82; 94; 229
 - generate 83; 112; 129; 138
 - produce 84
 - test 82; 229
 - types of 82
- every*
 - every <description>** 143
- examining program files 254
- except*
 - except <term>** 125
- execute actions
 - go* and *call* 89
- execute*
 - execute a procedure** 217
 - execute cyclically.** 72
 - execute randomly.** 72
 - execute sequentially.** 72
- execution monitor 79
- execution monitor declaration 72
- exiting LISP (see *lisp*) 64

- exiting ROSIE (see *logout*) 58
- exponential notation 20
- exponentiation operator (^ and **) 164
- extended string syntax 23
- external access 243
- false*
 - a proposition is false [in a database]* 214; 238
 - a proposition is provably false* 213; 237
- file items 20
- file package 253
- file specifiers 199
- filename*
 - an element is a filename* 247
- files (see *program files*) 253
- filesegment*
 - an element is a filesegment* 154
 - the filesegment from a string* 175
- filesegments 199
 - application of 257
 - operations on 260
 - rule sequence specifiers 258
 - shorthand notation 199; 259
- files
 - loading LISP files (see *dskin*) 64
- find*
 - find a string in a filespec* 265
- first member*
 - the first member of a tuple* 168
- fix*
 - fix [a line]* 60
- fixed format strings 171
- fixed format*
 - fixed format <subpat> [, <subpat>]** 183
- floor*
 - the floor of a number* 165
- for each*
 - for each <description>, <action block>* 101
- forget*
 - forget about an element [in a database]* 61; 234
- format*
 - fixed format <subpat> [, <subpat>]** 183
 - free format <subpat> [, <subpat>]** 182
- formfeed*
 - formfeed[s]* 190
 - page[s]* 190
- FORTUNE 285
- free format strings 171
- free format*
 - free format <subpat> [, <subpat>]** 182
- garbage collection (see *reclaim*) 65
- generate demons 82

- generate events 83; 112; 129; 138
- generator demons 83
- generator rulesets 76
- global database 221
- go 89
- greater
 - a number is greater than [or equal to] a number* 165
 - <term> is [not] greater than [or equal to] <term>* 111
- has
 - <term> has <a/an> <description>* 112
 - <term> has just one <description>* 112
 - <term> has more than one <description>* 112
 - <term> has no <description>* 112
- headers 69
- help
 - help!** 273
- HILEV 15; 64; 253
- history facility 44
- if
 - if <condition> <then part> [<else part>]* 96
- increment
 - increment a description [by a number] [in a database]* 210; 237
- info
 - info date* 63
 - info loaded* 63
 - info switches* 62; 342
 - info system* 63
- input/output 241
- input
 - a file is available for input* 248
 - a file is open for input* 248
 - a file is open for input/output* 248
 - the standard input channel* 247
 - open a file for input* 246
 - redirect input [to a file]* 247
- insert
 - insert after a filespec* 264
 - insert before a filespec* 264
- instance
 - an instance of a description [in a database]* 209; 236
 - an instance of an element* 155
 - an element is an instance of a description [in a database]* 210; 236
 - an element was an instance of a description [in a database]* 210; 236
 - an element will be an instance of a description [in a database]* 210; 236
- instantiate
 - instantiate a description to an element [in a database]* 209; 236
- integer
 - an integer from a lower bound to an upper bound [by a step]* 166
 - an element is a negative integer* 164
 - an element is a positive integer* 164
 - an element is an integer* 164

- intentional descriptions 207
 - call-by-name* property of 209
 - evaluation of 207
 - operations on 209
- intentional elements 149
 - class elements 201
 - intentional descriptions 207
 - intentional procedures 217
 - intentional propositions 211
- intentional procedures 217
 - operations on 217
- intentional propositions 211
 - evaluation of 211
 - operations on 212
- interrupts 56
- intransitivity 108
- iterative actions 91; 100
- iterative terms 145
- justify*
 - center justify** <subpat> [<dimen>] 184
 - CJ** [<term>] [**by** <term>] : <subpat> 184
 - left justify** <subpat> [<dimen>] 184
 - LJ** [<term>] [**by** <term>] : <subpat> 184
 - right justify** <subpat> [<dimen>] 184
 - RJ** [<term>] [**by** <term>] : <subpat> 184
- label*
 - the label of a number* 164
- labeled constant*
 - an element is a labeled constant* 164
- labeled constants 163
- LAMBDA 85
- lambda form 85
- last member*
 - the last member of a tuple* 168
- left justify*
 - left justify** <subpat> [<dimen>] 184
 - LJ** [<term>] [**by** <term>] : <subpat> 184
- length*
 - the length of a string* 174
 - the length of a tuple* 167
- less*
 - a number is less than [or equal to] a number* 165
 - <term> is [not] less than [or equal to] <term> 111
- let*
 - let** <term> **be the** <description> [**and** <term> **be the** <description>]* 94; 235
- letter*
 - [non]letter[s] [[not] in <term>] 186
- letters 19
- line*
 - line[s] 188
- lisp*

- lisp** 64
- LISP**
 - entering (see *lisp*) 64
 - exiting (see *lisp*) 64
 - garbage collecting (see *reclaim*) 65
 - loading files (see *dskin*) 64
- LIST-TO-TUPLE (LST)** 87
- list**
 - list a filespec** 262
 - list [(| ruleset | <integer> |)]!** 271
- load**
 - load a filespec** 260
- loading LISP files (see *dskin*) 64
- loading program files 254
- log**
 - the [anti]log of a number** 166
- logout** 66
- lowercase**
 - the lowercase of a string** 174
- match**
 - match a string against a pattern** 176; 196
 - match <term>: [selector[;]]* [default: <action block>[;]]** 98
- matched**
 - a string is matched by a pattern** 176; 197
- member**
 - a member of a tuple [from a position]** 167
 - the first member of a tuple** 168
 - the last member of a tuple** 168
 - the member of a tuple at a position** 168
 - the second member of a tuple** 168
- mixed format strings 171
- modifying program files 254; 256
- monitor rule 44
- move**
 - move a filespec after a filespec** 263
 - move a filespec before a filespec** 263
- multiplication operator (*) 164
- name**
 - an element is a name** 154
 - the name from a string** 175
- names 159
- negated propositions 110
- negated**
 - a proposition is negated** 214
- negation**
 - the negation of a number** 165
- negative integer**
 - an element is a negative integer** 164
- negative number**
 - an element is a negative number** 163

- NIL 87
- nonalphanumeric*
 - [non]alphanumeric[s] [[not] in <term>] 186
- nonblank*
 - [non]blank[s] [[not] in <term>] 186
- noncontrol*
 - [non]control[s] [[not] in <term>] 186
- nondigit*
 - [non]digit[s] [[not] in <term>] 186
- nonletter*
 - [non]letter[s] [[not] in <term>] 186
- nonnumber*
 - [non]number[s] [[not] in <term>] 186
- nonnumeral*
 - [non]numeral[s] [[not] in <term>] 186
- nonrecoverable errors 251
- not*
 - <term> <is/do aux> not [<a/an>] [<term>] [<pphrase>] 110
- notice*
 - notice a filespec 261
- noticing program files 254
- noun phrase specifiers 26
- number*
 - a number from a lower bound to an upper bound [by a step] 166
 - a random number from a lower bound to an upper bound 166
 - an element is a negative number 163
 - an element is a number 154; 163
 - an element is a positive number 163
 - an element is a simple number 164
 - the number from a string 175
 - [non]number[s] [[not] in <term>] 186
- numbers 161
 - arithmetic operators 163
 - comparisons 163
 - constraints on 163
 - labeled constants 163
 - operations on 163
 - simple numbers 161
 - types of 161
 - unit constants 162
- numeral*
 - [non]numeral[s] [[not] in <term>] 186
- numeric operators 135
- numeric tokens 20
- numeric value*
 - the numeric value of a number 165
- one of*
 - one of <term> [, <term>]* [,] or <term> 146
- open*
 - a file is open for input 248
 - a file is open for input/output 248

- a file is open for output* 248
- an open channel* 247
- open a file for input* 246
- open a file for output* 246
- open a file to read* 246
- open a file to write* 246
- opening I/O channels 241
- or*
 - <conjunct> **or** <disjunct> 105
 - <disjunct> [, **or** <disjunct>]* 105
- OS channel 243
- OS channel*
 - the OS channel* 247
- output*
 - a file is open for input/output* 248
 - a file is open for output* 248
 - the standard output channel* 247
 - open a file for output* 246
 - redirect output [to a file]* 247
- overlay*
 - overlay** <subpat> **on** <subpat> [<coords>] [<padding>] 185
- pad*
 - pad** <subpat> 183
- page*
 - formfeed[s]** 190
 - page[s]** 190
- paraphrase*
 - paraphrase** 64
- parse tree generation 38
- parse*
 - parse a file** 262
- parsemode*
 - parsemode** 64
- parsing 15
- parsing errors 313
- pattern variables 191
- pattern*
 - an element is a pattern* 154
 - the pattern from a string* 175
- patterns 177
 - example application 195
 - operations on 196
 - outputting text 244
 - reading against 244
 - relations to strings 172
 - subpatterns 179
 - text formatting 177
 - text matching 178
 - the matching process 193
- physical databases 219
- POIROT 295

*pop***pop!** 273*positive integer**an element is a positive integer* 164*positive number**an element is a positive number* 163*precedence* 38

of arithmetic operators 137

of conditions 106

of unit constants 162

predicate rulesets 74*predication* 108*prepositional phrase* 39*prepositions* 26*primitive sentences* 107*print***print a name as a string** 176; 249**print a string [on a file]** 176; 248*private class declaration* 72*private classes* 71*private database* 78; 221*private***private:** <formal> [([initially] <term>)] [, . . .]*. 72**private?** 273*procedural demons* 82*procedural rulesets* 73*procedure**an element is a procedure* 155*the procedure from a string* 175*procedures* 90; 93*produce demons* 83*produce events* 84*produce***produce an element** 80**produce an element!** 272**produce an element.** 273*profile***profile report** 280**profile reset** 280**profile [a filespec]** 280*profiling facility* 278*program files* 253

building 254

compiling 254

disabling 254

editing 254; 256

enabling 254

examining 254

loading 254

modifying 254; 256

noticing 254

- proposition*
 - an affirmed proposition [from a database]* 212; 234
 - an element is a proposition* 155
 - the proposition from a string* 175
- propositions* 107
 - asserting* 228
 - denying* 228
 - testing* 228
- provably*
 - a proposition is provably false* 213; 237
 - a proposition is provably true* 213; 237
- quantified descriptive terms* 141
- queries* 22
- query*
 - the query from a proposition* 215
- quit*
 - quit [because a string]* 81; 251
 - quit!* 273
 - quit.* 273
- quote*
 - quote[s]* 189
- random monitor* 72
- random number*
 - a random number from a lower bound to an upper bound* 166
- range*
 - a number does range from a lower bound to an upper bound* 165
- read*
 - open a file to read* 246
 - read a pattern [from a file]* 196; 249
- reclaim*
 - reclaim* 65
- recoverable errors* 251
- redirect*
 - redirect input [to a file]* 247
 - redirect output [to a file]* 247
- redirecting standard I/O* 242
- redo*
 - redo* 60
 - redo a line [thru a line] [for N times]* 60
- relative clause specifiers* 26
- relative clauses* 120
- remove*
 - remove a proposition from a database* 213; 235
- rename*
 - rename a file to a file* 250
- reserved words* 26
- restore*
 - restore a file [to a database]* 234
- result*
 - result!* 271
- resume*

- resume** [(| **ruleset** | <integer>)]! 271
- return**
 - CR[s]** 189
 - return** 80
 - return!** 271
 - return[s]** 189
- reverse**
 - the reverse of a tuple* 169
- right justify**
 - right justify** <subpat> [<dimen>] 184
 - RJ** [<term>] [**by** <term>] : <subpat> 184
- RITA** 9
- root names** 116
- root**
 - the square root of a number* 166
- rule sequence specifiers** 199; 258
- rule variables** 126; 144
- rules** 21; 67
- ruleset headers** 21
 - in filesegment* 199
- rulesets** 69
 - generator** 76
 - predicate** 74
 - procedural** 73
 - system** 85
 - types of** 73
- runtime errors** 317
- save**
 - save as a file** 65
 - save** [*a filespec*] 264
- scan**
 - scan a filespec** 262
- scanning errors** 313
- scanning program files** 254
- second member**
 - the second member of a tuple* 168
- select**
 - select** <term>: [**selector**[:]]* [**default**: <action block>[:]] 97
- send**
 - send a string** [*to a file*] 175; 248
- sentences** 106
 - propositions** 107
 - special forms** 110
 - types of** 106
- separator characters** 18
- sequential monitor** 72
- set**
 - a switch is set* 62; 342
- show**
 - show** [*a database*] 233
 - the descriptive terms* 138

- simple elements 149
 - filesegments 199
 - labeled constants 163
 - names 159
 - numbers 161
 - patterns 177
 - simple numbers 161
 - strings 171
 - tuples 167
 - unit constants 162
- simple number*
 - an element is a simple number* 164
- simple numbers 161
- sine*
 - the [arc]sine of a number [in radians]* 166
- some*
 - some** <description> 142
- something*
 - something** 188
- sort*
 - sort a tuple in ascending order** 169
 - sort a tuple in ascending pair order** 169
 - sort a tuple in descending order** 169
 - sort a tuple in descending pair order** 169
- special sentence forms 110
- square root*
 - the square root of a number* 166
- square*
 - the square of a number* 166
- standard input*
 - the standard input channel* 247
- standard output*
 - the standard output channel* 247
- stop*
 - stop dribbling** 61; 250
- string delimiter 18
- string tokens 20
- string*
 - an element is a string* 154
 - the string from an element* 174
- strings 171
 - extended syntax 23
 - fixed format 171
 - format attribute 171
 - free format 171
 - mixed format 171
 - operations on 173
 - relations to patterns 172
- subpatterns 179
 - conjunction of 181
 - disjunction of 182

- text formatting 182
- text matching 186
- <integer> or (| more | less | fewer |) [of] <subpat> 186
- <integer> [of] <subpat> 189
- <term> 188
- adjoin <subpat> [, <subpat>]* 185
- anything 187
- backspace[s] 190
- blank[s] 190
- box <subpat> to width <term> 183
- bs 190
- center justify <subpat> [<dimen>] 184
- character[s] [[not] in <term>] 186
- charcode <term> 191
- CJ [<term>] [by <term>] : <subpat> 184
- codes (<integer> [, <integer>]*) 189
- control <term> 191
- CR[s] 189
- end 190
- EOL[s] 190
- esc 190
- escape[s] 190
- fixed format <subpat> [, <subpat>]* 183
- formfeed[s] 190
- free format <subpat> [, <subpat>]* 182
- left justify <subpat> [<dimen>] 184
- line[s] 188
- LJ [<term>] [by <term>] : <subpat> 184
- overlay <subpat> on <subpat> [<coords>] [<padding>] 185
- pad <subpat> 183
- page[s] 190
- quote[s] 189
- return[s] 189
- right justify <subpat> [<dimen>] 184
- RJ [<term>] [by <term>] : <subpat> 184
- something 188
- tab[s] 190
- [non]alphanumeric[s] [[not] in <term>] 186
- [non]blank[s] [[not] in <term>] 186
- [non]control[s] [[not] in <term>] 186
- [non]digit[s] [[not] in <term>] 186
- [non]letter[s] [[not] in <term>] 186
- [non]number[s] [[not] in <term>] 186
- [non]numeral[s] [[not] in <term>] 186
- { <subpat> [, <subpat>]* } 181
- { <subpat> [| <subpat>]* } 182
- substitute*
- substitute** *an element for an element in an element* 156
- substitution*
- the substitution of an element for an element in an element** 156
- subtraction operator (-) 164

- such that*
 - (**such that** <condition>) 122
 - such that** (<condition>) 122
 - such that** <sentence> 122
- such*
 - such** [<a/an>] <class noun> [(<desc var>)] 127
- suspending ROSIE 58
- swap*
 - swap in a database** 232
- switch*
 - switch off a switch** 61; 342
 - switch on a switch** 61; 342
- sysload*
 - sysload a filespec** 261
- sysouts (see *save*) 65
- system ruleset library 87
- system rulesets 85
- system switches 58; 337
 - operations on 342
- system*
 - system ruleset <header>:** 85
- tab*
 - tab to an integer [on a file]** 248
 - tab[s]** 190
- tail*
 - a tail of a tuple [from a position]** 168
 - the tail of a tuple at a position** 168
- tangent*
 - the [arc]tangent of a number [in radians]** 166
- terminator characters 19
- terms 133
 - anaphoric 144
 - arithmetic 135
 - descriptive 137
 - elementary 134; 149
 - iterative 145
 - quantified descriptive 141
 - rule variable 144
 - simple descriptive 138
 - types of 133
- test demons 82
- test events 82; 229
- testing propositions 228
- text formatting 177
- text formatting subpatterns
 - <integer> [of] <subpat> 189
 - <term> 188
 - adjoin <subpat> [, <subpat>]* 185
 - backspace[s] 190
 - blank[s] 190
 - box <subpat> to width <term> 183

- bs 190
- center justify <subpat> [<dimen>] 184
- charcode <term> 191
- CJ [<term>] [by <term>] : <subpat> 184
- codes (<integer> [, <integer>]*) 189
- control <term> 191
- CR[s] 189
- end 190
- EOL[s] 190
- esc 190
- escape[s] 190
- fixed format <subpat> [, <subpat>]* 183
- formfeed[s] 190
- free format <subpat> [, <subpat>]* 182
- left justify <subpat> [<dimen>] 184
- LJ [<term>] [by <term>] : <subpat> 184
- overlay <subpat> on <subpat> [<coords>] [<padding>] 185
- pad <subpat> 183
- page[s] 190
- quote[s] 189
- return[s] 189
- right justify <subpat> [<dimen>] 184
- RJ [<term>] [by <term>] : <subpat> 184
- tab[s] 190
- { <subpat> [, <subpat>]* } 181
- text matching 178
 - the matching process 193
- text matching subpatterns
 - <integer> or (| more | less | fewer |) [of] <subpat> 186
 - <integer> [of] <subpat> 189
 - <term> 188
 - anything 187
 - backspace[s] 190
 - blank[s] 190
 - bs 190
 - character[s] [[not] in <term>] 186
 - charcode <term> 191
 - codes (<integer> [, <integer>]*) 189
 - control <term> 191
 - CR[s] 189
 - end 190
 - EOL[s] 190
 - esc 190
 - escape[s] 190
 - formfeed[s] 190
 - line[s] 188
 - page[s] 190
 - quote[s] 189
 - return[s] 189
 - something 188
 - tab[s] 190

- [non]alphanumeric[s] [[not] in <term>] 186
- [non]blank[s] [[not] in <term>] 186
- [non]control[s] [[not] in <term>] 186
- [non]digit[s] [[not] in <term>] 186
- [non]letter[s] [[not] in <term>] 186
- [non]number[s] [[not] in <term>] 186
- [non]numeral[s] [[not] in <term>] 186
- { <subpat> [, <subpat>]* } 181
- { <subpat> [| <subpat>]* } 182
- that*
 - that** (| <verb phrase> | <special vp> |) 123
 - that** <class noun> 145
 - that** <term> (| <verb phrase> | <rel op> |) 124
- the*
 - <term> ' s <description> 138
 - the** <description> 138
- there*
 - there is** <a/an> <description> 112
 - there is just one** <description> 112
 - there is more than one** <description> 112
 - there is no** <description> 112
- thing*
 - an element is a thing* 153
- to*
 - to** <atom> [<formal>] [<private pp>]*: 73
 - to decide** <formal> <be aux> <a/an> <root name> [<private pp>]*: 74
 - to decide** <formal> <be aux> <atom> [<formal>] [<private pp>]*: 74
 - to decide** <formal> <be aux> <prep> [<formal>] [<private pp>]*: 74
 - to decide** <formal> <do aux> <atom> [<formal>] [<private pp>]*: 74
 - to generate** [(| **the** | **a** | **an** |)] <root name> [<private pp>]*: 76
- toggle*
 - toggle** *a switch* 62; 342
 - toggle off** *a switch* 62; 342
 - toggle on** *a switch* 62; 342
- tokenization 17
- tokenization errors 313
- tokenizer 17
- tokens 20
- top-level monitor 43
- top*
 - top!** 273
- trace*
 - trace** [*a filespec*] 280
 - trace!** 272
- tracing execution 268
- tracing performance 278
- transcript files 245
- transitivity 108
- true*
 - a proposition is provably true* 213; 237
 - a proposition is true [in a database]* 214; 238

TTY channel

the TTY channel 247

tuple

an element is a tuple 154

the tuple containing each <description> 169

the tuple from a string 175

tuples 167

operations on 167

type

type a file 63; 249

unbreak

unbreak [a filespec] 280

unbreaking broken rulesets 279

unit constant

an element is a unit constant 164

unit constants 162

units

the units of a number 164

unless

unless <condition> <then part> [<else part>] 96

unload

unload a filespec 265

unnotice

unnotice a filespec 265

until

until <condition>, <action block> 101

untrace

untrace [a filespec] 280

up

up! 272

uppercase

the uppercase of a string 174

value

the absolute value of a number 165

the numeric value of a number 165

verb phrases 108

virtual database 224

virtual relations 226

where

(where <condition>) 122

where (<condition>) 122

where <sentence> 122

which

<prep> which <term> <verb phrase> 124

which (| <verb phrase> | <special vp> |) 123

which <term> (| <verb phrase> | <rel op> |) 124

while

while <condition>, <action block> 101

who

who (| <verb phrase> | <special vp> |) 123

who <term> (| <verb phrase> | <rel op> |) 124

whom

<prep> whom <term> <verb phrase> 124

whose

whose <description> <be aux> <term> 124

write

open a file to write 246